Valentin Gritsch, BSc.

An FPGA based custom waveform generator for Time-of-Flight measurements

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Electrical Engineering

submitted to

Graz University of Technology

Supervisor:

Univ.-Prof. Mag.rer.nat. Dr.rer.nat. Alexander Bergmann

Co-Supervisor:

Dipl.-Ing. BSc. Reinhard Klambauer

Institute of Electrical Measurement and Sensor Systems
Head: Univ.-Prof. Mag.rer.nat. Dr.rer.nat. Alexander Bergmann

Graz, February 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Acknowledgement

I would like to sincerely thank my two supervisors Reinhard Klambauer and Alexander Bergmann for giving me the opportunity to work on this challenging and exciting project.

In particular I would like to thank Reinhard Klambauer for his support on various topics as well as his preliminary work, making this thesis possible.

Finally I would like to express my dearest gratitude to my parents, all other relatives and friends, who have supported me for all my life. Special thanks goes to my mother, Annemarie Karner, who is responsible for my knowledge of the English language and has greatly helped me by proof reading this text.

Abstract

Prior to the project there was an existing Ultra Sonic Flowmeter Application at the Institute of Electrical Measurements and Sensor Systems (EMS), developed by Reinhard Klambauer as a part of his doctoral thesis, capable of generating various sinusoidal signals on 16 individual output channels. This shall be possible in the future and additionally the system shall be extended to generating various arbitrary shapes.

The existing device for flow measurement [20][31] is very capable of being used for Time of Flight (TOF) measurements described in the Austrian Research Promotion Agency (FFG) project Valerie [27][26] and is essential for MOGLI. Hence this thesis, as part of project Mogli, also describes in its first part, the theoretical foundations necessary to plan and implement the project. The project flow for hardware development and the individual hardware modules used are described. The design process of new hardware modules in Very High Speed Integrated Circuit Hardware Description Language (VHDL), as well as the implementation and communication of the control software are elaborated.

In the second part the implementation and hardware modules themselves are described. The necessary steps to generate a working development environment for potential other new features are elaborated. The implementation on the software side of the project in PetaLinux is defined and the functions used to control the hardware and the communication to the hardware are depicted.

At last the specific measurements with simple both and more complex signals are performed and the functionality verified. Their results are discussed to finally draw a conclusion. Concluding potential improvements of the system are outlined to bring in further research on implementation.

Zusammenfassung

Vor Beginn des Projekts konnte das bestehende Ultra Sonic Flowmeter des Instituts für elektrische Messtechnik und Sensorik, das im Zuge der Dissertation von Reinhard Klambauer entwickelt wurde, auf 16 individuellen Ausgängen sinusförmige Signale erzeugen. Dies soll weiterhin möglich sein und zusätzlich sollen willkürliche Signalformen erzeugt werden können.

Wie bereits im FFG Projekt Valerie [27][26] gezeigt wurde ist das bestehende Flowmeter [20][31] auch sehr gut geeignet um TOF Messungen durchzuführen. Die folgende Arbeit beschreibt als Teil des Projekt Mogli, das auf Valerie aufbaut, im ersten Teil die notwendigen theoretischen Grundlagen für die Planung und Umsetzung des Projekts. Es wird der verwendete Implementierungsablauf bei Hardwareentwicklungen beschrieben und die einzelnen Hardwarebausteine erklärt. Der Entwurfsprozess von eigenen Hardwarebausteinen in VHDL wird dargelegt, sowie die Implementierung der Steuersoftware und die Kommunikation zwischen ebenjener Steuersoftware und der Implementierung erklärt.

Im folgenden Teil wird die konkrete Implementierung der Hardware beschrieben und die Konfiguration der einzelnen Bausteine erklärt. Die notwendigen Schritte zur Generierung einer Entwurfsumgebung für weitere potentielle Erweiterungen sowie die Nutzung der Software wird beschrieben. Weiters wird die softwareseitige Implementierung in PetaLinux definiert und beschrieben. Die Funktionen zur Steuerung der Hardware und der Kommunikation werden gezeigt und erklärt.

Zu guter Letzt werden Messungen sowohl mit einfachen als auch mit komplexeren Signalen durchgeführt und die Funktionalität verifiziert. Die Ergebnisse wurden diskutiert und eine Folgerung gezogen. Abschließend wird noch auf potentielle Verbesserungen eingegangen, die durch weitere Bearbeiter implementiert werden können.

Contents

Abstract iii

1 Introduction 1

2 Theoretical Foundations 3
2.1 Physical Foundations 3
2.1.1 Time-of-flight measurements 3
2.1.2 Mechanical Properties of Lithium Ion Batteries 6
2.2 Hardware 7
2.2.1 Embedded Systems 7
2.2.2 ARM®Architecture 7
2.2.3 Programmable Logic circuits 8
2.2.3.1 Field Programmable Gate Array 8
2.2.4 Zynq System-on-Chip 9
2.2.5 Direct Memory Access Engine 9
2.2.5.1 Descriptors 10
2.3 Packaging with Linux 11
2.3.1 Linux Distributions 11
2.3.2 Linux Shells 12
2.3.2.1 Keyboard Shortcuts 12
2.3.2.2 Relative versus absolute paths 13
2.3.2.3 Shell Variables 13
2.3.2.4 Shell Scripts 13
2.3.2.5 Help argument 14
2.3.2.6 history 14
2.3.2.7 grep 14
2.3.2.8 cd 14
2.3.2.9 ls 14
2.3.2.10 rm 15

2.3.2.11 mkdir 15
2.3.2.12 source 15
2.3.2.13 echo 15
2.3.2.14 ssh-keygen 16
2.3.2.15 if 16
2.3.2.16 ssh 16
2.3.2.17 chmod 16
2.3.2.18 sed 17
2.4 Embedded Software Development in Linux 17
2.4.1 Digital Logic Design 17
2.4.1.1 Vivado Design Flow 19
2.4.2 Vivado IP Packager 20
2.4.3 Hardware Description Languages - VHDL 21
2.4.3.1 VHDL Entities 21
2.4.3.2 VHDL Architectures 21
2.4.4 Advanced eXtensible Interface 22
2.4.5 Device Tree 23
2.4.5.1 DMA implementation 23
2.4.5.2 Memory Reservation 24
2.5 Server-Client Interfaces 25
2.5.1 QT Applications for ARM Architectures 26
2.5.2 Memory Mapping and Interfacing 26
2.5.3 QT Applications for Desktop Architectures 27
2.5.4 Communication Standard 27

3 Implementation **28**
3.1 Hardware 28
3.1.1 Xilinx Zynq®-7000 System-on-Chip 29
3.1.2 Carrier Board 29
3.2 Packaging 30
3.2.1 Shell Scripting and Automation 30
3.2.1.1 Installation of QT for cross-compiling for ARM® 30
3.2.1.2 Cable Drivers for Platform Cable 33
3.2.2 Complementary Scripts 33
3.2.2.1 Matlab Script to generate an output window 33
3.2.2.2 Other Scripts 34

3.3 Embedded Software - Xilinx . . . 35
3.3.1 Device Tree . . . 35
3.3.2 Address Map Layout . . . 35
3.3.3 Vivado IPs . . . 36
3.3.3.1 AXI Direct Memory Access . . . 36
3.3.3.2 AXI4-Stream Data FIFO . . . 39
3.3.3.3 AXI Interconnect . . . 40
3.3.3.4 AXI GPIO . . . 40
3.3.4 VHDL Implementation . . . 40
3.3.4.1 Delay Generator . . . 41
3.3.4.2 Output Switch . . . 43
3.3.5 Updating PetaLinux . . . 44
3.4 Server-Client Interfaces . . . 45
3.4.1 FlowmeterControl Application . . . 45
3.4.2 FlowmeterServer Application . . . 46
3.4.2.1 DMAHandler::setNewWindow . . . 46
3.4.2.2 DMAHandler::clearWindowMemory . . . 47
3.4.2.3 DMAHandler::setWindow . . . 47
3.4.2.4 DMAHandler::setMemorySize . . . 47
3.4.2.5 DMA_WINDOW::reset . . . 48
3.4.2.6 DMA_WINDOW::init . . . 48
3.4.2.7 DMA_WINDOW::start . . . 48
3.4.2.8 DMA_WINDOW::writeAllDescriptors . . . 48
3.4.2.9 WINDOW::init . . . 49
3.4.2.10 WINDOW::setDelayChannel . . . 49
3.4.3 Communication Standard . . . 49
3.4.3.1 CCmdSetDelay . . . 49
3.4.3.2 CCmdSetWindow . . . 50

4 Results **51**

5 Conclusion **57**

6 Outlook **58**

Acronyms **60**

Bibliography **62**

List of Figures

2.1 Dispersion diagram adapted from Peter Hadley [1] 4
2.2 Example of a pulse: five periods of a sine wave windowed with one cosine window 5
2.3 Output Voltage vs SoC curve for a Li-ion battery by Habiballah Rahimi Eichi et al. [30] 5
2.4 Layout of a programmable hardware in general by Florijan Reichmann [31] 9
2.5 Block diagram of memory access with a DMA Controller . . . 10
2.6 Block diagram of memory descriptors 11
2.7 Vivado Project Flow 18
2.8 Device Tree graphical representation 23

3.1 Block diagram of a measurement flow 28
3.2 Schematic of the Zynq board with paths for ultrasonic signal generation and sensing by R. Klambauer, A. Bergmann [20] . 29
3.3 Block diagram hardware 38
3.4 Graphical User Interface (GUI) representation in Flowmeter-Control 45
3.5 Block representation of the software functions after a command is received on the server 46

4.1 Settings for test signal 51
4.2 Resulting signal (multiple pulses) 52
4.3 One pulse of the resulting signal 52
4.4 Settings for two outputs with different signals 53
4.5 Two outputs with different signals 53
4.6 Settings for five periods of a sinusoidal signal multiplied by one cosine window 54

4.7 One pulse of five periods of a sinusoidal signal multiplied by one cosine window 54
4.8 FFT of one pulse of five periods of a sinusoidal signal multiplied by one cosine window in blue and only the same 5 sinusoidal signals without windowing in red up to 50 MHz . 55
4.9 FFT of one pulse of five periods of a sinusoidal signal multiplied by one cosine window in blue and only the same 5 sinusoidal signals without windowing in red up to 5 MHz . . 55
4.10 FFT of one pulse of five periods of a sinusoidal signal multiplied by one cosine window in blue and only the same 5 sinusoidal signals without windowing in red up to 1 MHz . . 56

1 Introduction

This Master's Thesis uses the Master's Thesis "Embedded System Design for a Time-of-Flight Ultrasonic Flowmeter" of Florijan Reichmann, BSc., MSc. [31]. as a baseline and follows up on its achievements. The result of the above thesis has been extended to another useful part: the generation of custom waveforms. Beforehand the EMS had developed an ultrasonic measurement alternative to determine various flows as described in Section 2.1.1 In principle the system generates a sine wave with a predefined frequency, amplitude and phase. This wave can be sent into a piezoelectric sensor, for example, and then read in with the same module.

Additionally, this thesis is part of the Mogli project in cooperation with the Austrian Institute of Technology (AIT) and the company partners TDK Corporation (TDK) and Anstalt für Verbrennungskraftmaschinen List (AVL). The aim of this project is to improve the laboratory proven method of ultrasound diagnostics for lithium-ion accumulators such that it becomes viable in practical applications building on the FFG project Valerie (#865 148).

The main motivation for the Mogli project and hence also for this thesis is to get the European Union (EU) emission goals for the year 2030 further into reach by decreasing the emissions of the transporting sector. This goal can only be reached by improving existing energy storage solutions, in this case lithium-ion batteries. By developing this innovative and cost-effective measurement application, the safety and lifespan of the lithium-ion batteries shall be increased.

The main part of the System is a PicoZed System-on-Module (SoM) board housing a Xilinx Zynq 7020 System-on-Chip, accompanied by a custom Printed Circuit Board (PCB), the so-called carrier board. This board provides Analog to Digital Converters (ADCs), as well as Digital to Analog Converters (DACs) and their respective necessary components. The Zynq board is

running on PetaLinux, an embedded-Linux operation system controlling the Field Programmable Gate Array (FPGA) via a certain user space application, as seen in [31]. Before the initiation of this project signal generation was already working. That includes the generation of sinusoidal output signals of frequencies up to 300 kHz, with an amplitude of up to 2.5 V on 16 independent channels.

In addition to keeping all working parts that previously existed as untouched as possible and in any case fully working, it was intended and achieved to generate custom signals, based on previously generated values, for example in Matlab. Those values should be imported into the software and written to the memory of the embedded system. From there the individual periods of the signal shall be played onto the outputs. One shall be able to define a pause in between the single periods, to be able to measure and observe the reaction of any device under test. There should also be as many different channels as possible. First the theoretical foundations have been characterized, next the implementation is being described in detail. At last the results are portrayed and a conclusion drawn.

2 Theoretical Foundations

The underlying principles needed for the implementation of this Flowmeter are covered in this part on the theoretical foundations. This part is divided into five main categories: Physical Foundations, Hardware, Packaging, Embedded Software Development in Linux and Server-Client Interfaces.

2.1 Physical Foundations

In this short chapter on physical foundations, in particular, on TOF measurements, in general, the reader is educated on why such measurements are a good method of non-destructive testing in various applications.

2.1.1 Time-of-flight measurements

In general, a TOF measurement is used to determine the distance of an object with respect to the sensor. To achieve this one needs to emit a signal (for example electrically or acoustically), wait for it to reflect off the surface of an object and then measure the signal again. The time passing from emission and arrival of the signal is proportional to the distance.

The device used, described in Section 3.1.2 has been intended to prove the feasibility of a new measurement principle for Flowmeters in harsh environments, for instance estimating mass emissions of combustion sources [20]. As thoroughly explained by Florijan Reichmann [31], Flowmeters are devices to measure the flow of volume or mass in an enclosed volume. The device can also be used to implement a TOF Flowmeter.

It is from distance sensing capability that various applications for TOF measurements can be derived. First of all, one can place the sensor on top of a tank, facing downwards and hence monitor the fill or stock level in the tank. Additionally, one can use the precise measurement capability to non-destructively test welding points for cracks, applying the principle of Time of flight diffraction ultrasonics (TOFD), where a sensor and an actuator are placed on opposite sides of a weld. The sensor is then able to measure the incoming wave, being the sum of the wave travelling on the surface and the one being reflected off the wall. If there were a crack there would be a third wave, because diffraction appears on the tip of the crack, the depth of which can be calculated using trigonometry [19].

When choosing the signal for TOF-applications, one should especially consider that there are as little high frequency overtones as possible to stimulate only the first mode and receive a well-defined signal at the receiver. For low frequencies there is a linear relationship between the stimulating frequency and speed of the wave. As higher frequencies occur, their relationship is described by the dispersion relation [1]. Without going too much into detail, the relationship can be seen in Figure 2.1.

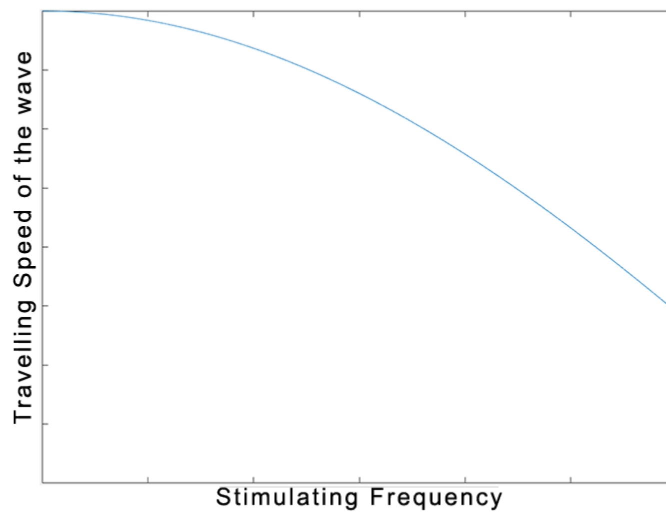

Figure 2.1: Dispersion diagram adapted from Peter Hadley [1]

To achieve this, one should not start a signal at once, but rather multiply the

required signal by a window and send the result. As described in [29] there are many possible windows, each with different advantages. For simplicity a cosine window has been used in this implementation, but can be replaced with any other one. An example for a possible pulse, in this case five periods of a sine wave windowed with one cosine window can be seen in Figure 2.2.

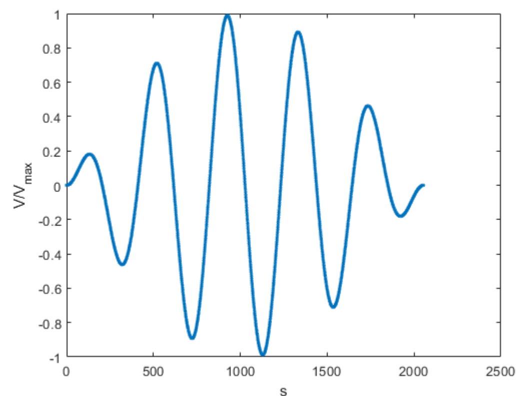

Figure 2.2: Example of a pulse: five periods of a sine wave windowed with one cosine window

When measuring the State of Charge (SoC) of Lithium Ion (Li-ion) batteries using their terminal voltage, one is presented with a very complex task, as the open-circuit voltage of the battery is very flat, as can be seen by the discharge curve in Figure 2.3. Hence more information, for example on mechanical properties are of use.

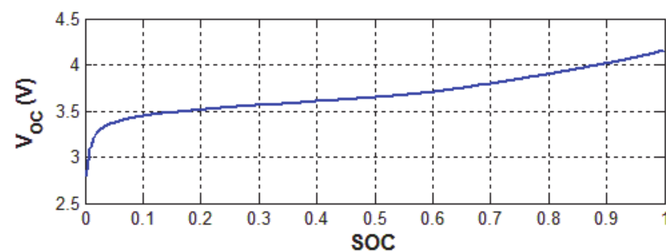

Figure 2.3: Output Voltage vs SoC curve for a Li-ion battery by Habiballah Rahimi Eichi et al. [30]

2.1.2 Mechanical Properties of Lithium Ion Batteries

There are various properties of a Li-ion battery of interest and you can say that the more properties you know the better. Usually the cell voltage and the output current are measured and the SoC is estimated. Adding mechanical properties to the mix will improve characterizing the battery.

There are three key states in Li-ion batteries, State of Fitness (SoF) and State of Health (SoH) and SoC of the cell. The latter describes on a scale from 0 to 100 % how much charge there is in the battery. The SoH compares the condition of the battery to a brand-new one and the SoF describes if the battery is currently performing as expected.

Hartmut Popp et al. [26] state

> *During battery operation, the lithiation level of the anode, repesentative of SoC, has a significant impact on cell behaviour, ageing, and cyclability. Because of the complex physico-chemical nature of the lithium-ion battery, identifying the internal changes that lead to battery degradation and failure is challenging, but as a feature of interest extracted from a non-destructive ultrasonic response signal, TOF could be used for analysing battery performance, which leads to changes in both mechanical impedance and ultrasonic velocity.*

The underlying principle can also be applied to other measurement applications. Purim Ladpli and Fotis Kopsaftopoulos and Fu-Kuo Chang [21] describe in their work the possibility of using a piezoelectric actuator and a corresponding sensor on a Li-ion Battery to estimate the SoC and SoH of the battery under test. It shows the feasibility of this measurement principle for the battery measurements. One major advantage of this method is that existing battery packs can be retrofitted with sensors.

As researched by Ladpli et. al [21] and improved on by Hartmut Popp et al. [28] it is possible to measure the travelling time of the wave generated by a piezoelectric actuator and estimate SoC and SoH by this measured time.

2.2 Hardware

This Section is intended to showcase the used hardware modules of the Xilinx Zynq 7020 System-on-Chip board and the carrier board.

2.2.1 Embedded Systems

As seen in the embedded systems glossary [5] embedded systems take up the majority of produced processors worldwide. In general, they are the brains of every modern electronic device. The size of embedded systems may vary from the smallest 4 bit micro-controllers up to the very powerful Digital Signal Processor (DSP) In general, an embedded system is a combination of computer hardware and software performing one or more dedicated functions. One main advantage of an embedded system is that all necessary parts, in general Central Processing Unit (CPU), Random Access Memory (RAM), storage and basic peripherals are integrated together, either on one board (single board unit) or on one single chip altogether. The latter would be called Microcontroller (MCU). There are various instruction sets on the basis of which embedded systems are built, the most common among which is the Advanced Reduced Instruction Set Computer Machine (ARM®) architecture.

2.2.2 ARM®Architecture

ARM® was launched in 1985 [5] as the first commercial Reduced Instruction Set Machine (RISC) processor. As can be seen from the Acorn-ARM® Story [4] ARM® are licensing out Intellectual Property (IP)-cores and not building their own hardware. According to ARM® Limited [22], the main advantage of ARM® compared to standard i86 processors, being a Complex Instruction Set Machine (CISC), is their very low power consumption and their low price which is due to the smaller number of general purpose instructions, accounting for a cheaper and more efficient silicon chip.

ARM® micro-controllers have been developed as a battery saving supplement to the power hungry i86 processors and have started to be considered

the standard, when an estimated 77 % of the embedded RISC market was ARM® technology, as described in the Acorn-ARM® story [4]. Regarding consumer electronics, they were mostly used in various devices up to mobile phones, but recent developments have shown first implementations of ARM® into laptops as well.

2.2.3 Programmable Logic circuits

Logic circuits, in general, are circuits consisting of various logic gates and Input and/or Output (I/O) ports connected in the desired fashion. Of course there are also custom logic circuits, produced for one specific application, for example the Application Specific Integrated Circuit (ASIC). However, since a very complicated and highly customized production process is required for them, their use makes sense for a very large number of devices, in the millions of pieces range. Hence programmable logic circuits are the preferred alternative to fixed logic ones.

There are various types of programmable logic, the most commonly used are, according to the embedded systems glossary [5], the Programmable Logic Device (PLD), the Complex Programmable Logic Devices (CPLDs) and the FPGA. As the relevant project has been implemented with the latter, solely the FPGA will be thoroughly explained here.

2.2.3.1 Field Programmable Gate Array

An FPGA is a very popular hardware module the internal connections of which can be easily reprogrammed. Any digital and Turing-complete hardware can be implemented in an FPGA as a prototype and later produced in bulk on a custom logic chip. As FPGAs contain a huge amount of possible interconnects, the main area of the chip is occupied by them and hence they are very expensive, compared to an ASIC for example. However, the advantages, for instance the ease to reprogram them and the fact that various debugging capabilities exist, overweigh at whilst prototyping. Other major advantages of FPGAs are their ability of real-time operation and the possibility to speed up calculations for parallel computing. Another reason

to choose FPGAs for implementation is brought in if only one or only a few devices performing a certain task are required. This is demonstrated within the scope of this project.

An FPGA consists of configurable I/O and logic blocks and an interconnection programmable network, as depicted in Figure 2.4. Each of the logic blocks can be a simple macrocell, consisting of combinatorial logic and a flip-flop, as they are seen in PLDs, or also a lot more complex.

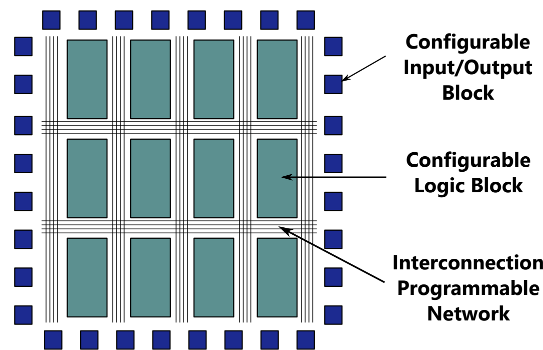

Figure 2.4: Layout of a programmable hardware in general by Florijan Reichmann [31]

2.2.4 Zynq System-on-Chip

The Zynq platform devices are fully programmable System-on-Chip solutions by Xilinx Inc. It is a one-chip implementation of a dual core ARM® and a 7-series FPGA by Xilinx Inc. The former is referred to as the Processing System (PS) and the latter is referred to as the Programmable Logic (PL). Those two main components are connected to each other by Advanced eXtensible Interface (AXI) connection. This system is usually programmed with Vivado, an Integrated Development Environment (IDE) provided by Xilinx Inc.

2.2.5 Direct Memory Access Engine

Usual memory access is only yielded by the processor. It performs its calculations and fetches and stores data from and to memory, whenever

necessary. This of course works for basic applications, but as more things need to be done simultaneously, or multiple programs are run, transferring huge amounts of data decreases overall performance. Additionally, the processor can only yield memory access to one program at a time, so while the processor is performing other non-memory related tasks, the memory bus remains unused.

Direct Memory Access (DMA), on the other hand, only requires minimal interaction of the processor. As seen in the embedded systems glossary [5], the DMA Controller utilizes the cycles when the processor is not accessing the memory lanes on the shared bus and transferring data on its own. This leads to the main advantage that the processor can perform whatever task while memory content is being transferred to the respective device by the DMA engine. A graphical representation of how such a transfer is performed can be seen in Figure 2.5.

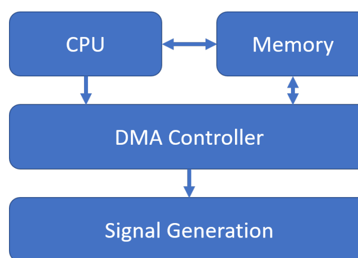

Figure 2.5: Block diagram of memory access with a DMA Controller

2.2.5.1 Descriptors

To be able to describe the location of data in memory, one uses memory descriptors. Each of them is 64 bit and describes on the one hand the location of the next descriptor and on the other hand a memory location variable. The memory location can have significantly more space, in this case 128 bit. To generate a descriptor chain for continuous transfer, one can set the last descriptor in a way that it point to the first one, as can be seen in Figure 2.6.

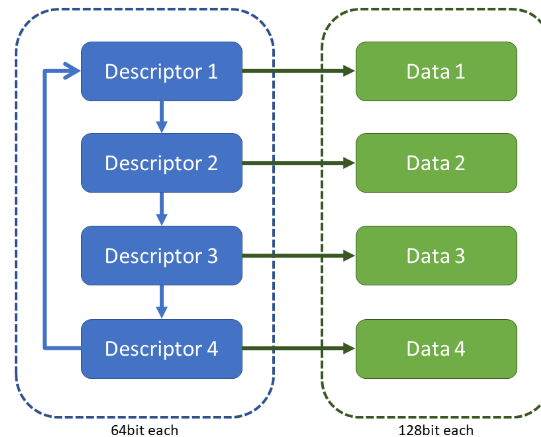

Figure 2.6: Block diagram of memory descriptors

2.3 Packaging with Linux

One of the major but often disregarded tasks within a project is packaging it as such that the next person working on it can start off with as little training time as possible. Also, minimizing any necessary recurring tasks is helpful. This short paragraph is a tribute to some of the most important features of packaging.

2.3.1 Linux Distributions

When comparing Linux Distributions to the windows operating systems, the main difference is that most Linux operating systems are free and that one can implement more easily recurring tasks by using shell scripts and commands, as described in Section 2.3.2. There are many Linux Distributions, each one of which is an operating system based on the Linux kernel and a package management system. Each of these distributions has many unique features. The most common ones are Debian and Ubuntu. Each of them come with various Desktop environments, if the graphical version is installed and their main difference is that Debian is completely free and utilizes as little proprietary software as possible, while Ubuntu focusses on

usability, including many more drivers and software within their default repository [23]. As Ubuntu is used within this project, the following parts will focus on it and especially its shell.

Ubuntu Linux, or in short, just Ubuntu is an open-source, unix based operating system published and maintained by Canonical Ltd. One of its advantages compared to other Linux operating systems is its capability of working out of the box and the vast amount of included free and non-free drivers and software.

2.3.2 Linux Shells

Most Linux Distributions contain many different shells, the most common one being *bash* [34]. It has used for automations in this project. Each shell has many different built-in commands, some of which will be elaborated here. Additionally, one can run programs from shell. The most commonly used programs and commands for this project are listed and described in this section.

One can pass arguments to the commands by adding them after the command, separated by a space. The result of a command can be piped onto another one, for example by using the "|" operator. There is always many ways to achieve the desired result and only one of them, certainly not always the best one, is elaborated here. A sample on how to use *command1* with *argument1* and send its output to *command2* can be seen in Listing 2.1. One can also use the wildcard "*" within many arguments for commands.

```
1 command1 argument1 | command2
```

Listing 2.1: Example of a Linux Shell command

2.3.2.1 Keyboard Shortcuts

The standard keyboard shortcuts can be extended by the user. There might be more shortcuts within shell programs, but the most commonly used are *Tab* to autocomplete the current command or folder, *Ctrl+D* to close the current session and *Ctrl+C* to terminate the current command execution.

2.3.2.2 Relative versus absolute paths

In Linux every file, folder and even disk is always accessible within the same root directory /, while in Windows every new drive is assigned a new drive-letter different from the one of the already existing ones, for example "D:/".

Every device, even hard disks are represented as files within the Linux path. When pointing towards the location of a file or folder, there are two different methods, either absolute or relative. The first of which always starts from the root directory /, the other is starting from the current directory. When being in */home/user1*, for example, one can access the root directory / by its absolute representation / or by the relative one *../../*, as *..* can be used to jump one directory up.

2.3.2.3 Shell Variables

One can use shell variables to define values once and use them when needed. To do this, one needs to place the variable's name on the left side of an equal sign and its value on the right side, as seen in Listing 2.2 Line 1. The variable can then be used within any command by using the code in Line 3 until the current shell session ends.

```
1 ZED_TCP_IP=192.168.178.113
2
3 $ZED_TCP_IP
```

Listing 2.2: Example of a shell variable

2.3.2.4 Shell Scripts

Shell commands can be placed within a text file which then can be run, meaning that the commands are executed subsequently. This can automate certain recurring tasks easily. To run a file, one needs to give it the corresponding permission as seen in Section 2.3.2.17 first and then can run it using *./path/filename*.

2.3.2.5 Help argument

Most commands provide some explanation on their arguments and usage, which, in many cases, can be displayed by adding the argument *h* or *–help*.

2.3.2.6 history

This essential command displays the whole command history of the user, which is commonly used to reproduce executed programs and commands and as rudimentary documentation.

2.3.2.7 grep

Grep may be used to search within files or within the output of other commands. To search for the word *Linux* within the output of *history*, as described in Section 2.3.2.6, one can use the example in Listing 2.3

```
1 history  grep "Linux"
```

Listing 2.3: Example of the grep Shell command

2.3.2.8 cd

The *cd* or change directory command with the desired directory as an argument can be used to switch to another directory.

2.3.2.9 ls

The *ls* command is used to display the content of the given directory. If no directory is given as an argument, it uses the current. The other most common arguments, given to this command after a '-' symbol, are *a*, *h* and *l*. The order of these arguments is not relevant to the result. In the given order, the first argument is used to show hidden files as well, the second one to display the results in a human readable form and the last one to list other information, such as the owner, the permissions and the size of the files and

directories displayed. To show the content of the */home/* directory, one can use the command in Listing 2.4.

```
1 ls -ahl /home/
```
Listing 2.4: Example of ls Shell command

2.3.2.10 rm

The *rm* command is used to remove a file, as seen in Listing 2.5 Line 1. When deleting a folder, the argument *-R* is needed, as seen in Line 2.

```
1 rm ../temp.csv
2 rm -R /temp/
```
Listing 2.5: Example of rm Shell command

2.3.2.11 mkdir

To create a new directory one can use the *mkdir* command with the path and name of the directory as an argument, as seen in Listing 2.6.

```
1 mkdir /home/user1/Temp
```
Listing 2.6: Example of mkdir Shell command

2.3.2.12 source

To load a shell file, for example a file containing variable definitions as seen in Section 2.3.2.3 one can use the source command, as seen in Listing 2.7.

```
1 source /home/user1/variables
```
Listing 2.7: Example of source Shell command

2.3.2.13 echo

To print text on the command line one can use the *echo* command, as seen in Listing 2.8.

```
1 echo "Working on important tasks"
```
Listing 2.8: Example of echo Shell command

2.3.2.14 ssh-keygen

The *ssh-keygen* program can be used to generate an *ssh key*, which can then be used to connect to other devices via the *ssh* protocol without a password, using key based authentication. To generate a standard key for the current user, one can use the command without any arguments.

2.3.2.15 if

The *if* statement can be used to test a condition and act accordingly. The full list of conditions can be seen in the corresponding bash guide [32]. An example to check if the file */home/user1/test* exists has been given in Listing 2.9.

```
1 if [ -a "/home/user1/test/" ]; then
2     echo "File exists."
3 fi
```

Listing 2.9: Example of the if statement in Shell commands

2.3.2.16 ssh

One can connect to another machine and execute a shell by using the *ssh* command. To connect to the machine with the IP-address *192.168.178.11* with the user *user1*, one can use the command in Listing 2.10. If an *ssh* key had been previously generated using Section 2.3.2.14, one can enable passwordless login on subsequent sessions by supplementing the *ssh* command with the *ssh-copy-id* command once.

```
1 ssh user1@192.168.178.11
```

Listing 2.10: Example of the if-statement in Shell commands

2.3.2.17 chmod

The *chmod* command can be used to change the permissions of a file or folder. In general, there are three different layers of permissions as well as three different permissions. The first layer would be the user layer, the

second one corresponds to a group of users and the last one everyone. Each layer can have individual permissions to read (*r*), write (*w*) and execute (*x*) the corresponding object. To add the permission to execute an object for every layer one can use the command in Listing 2.11. More information on this command can be found in [35].

```
1 chmod +x object
```

Listing 2.11: Example of chmod Shell commands

2.3.2.18 sed

The *sed* command can be used to replace text with other text. The example in Listing 2.12 can be used to display the command history, but replace the word "folder1" with the word "folder2".

```
1 history | sed 's/folder1/folder2/g'
```

Listing 2.12: Example of sed Shell commands

2.4 Embedded Software Development in Linux

This section intends to describe all the parts necessary to develop embedded software within a Linux environment for the Zynq Platform by Xilinx Inc. These parts range from utilizing existent modules to developing new ones, as far as the connection in between them and lead to the implementation of drivers to make them available to an operating system and use them within applications.

2.4.1 Digital Logic Design

In general, the design flow for digital logic is very straightforward and will be elaborated in the following sections. Further the method to generate custom IP is elaborated. In this development, a hardware description language (HDL) is used to describe the circuit. This serves, on the one hand, as a high level documentation and, on the other hand, can be converted into a lower level description of the circuit, which can then be fabricated. The

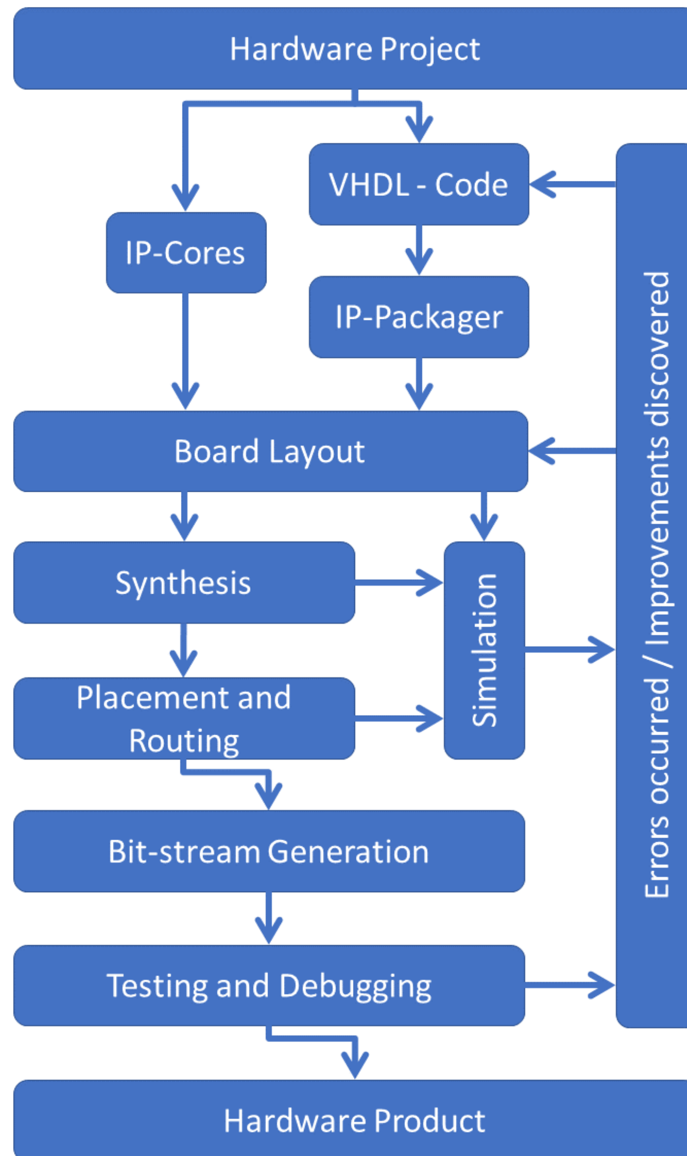

Figure 2.7: Vivado Project Flow

mainly used HDLs are VHDL, SystemC and Verilog, the former has been used for this project. At last the device-tree of Linux operating systems is described.

2.4.1.1 Vivado Design Flow

As in this special case a chip from Xilinx Inc. was used; their software Vivado is the preferred choice for designing and implementing its functionality. A graphical representation of the project flow can be seen in Figure 2.7. The design flow starts with high level description, in this case assisted by many IPs which may be added to the design with a mere drag and drop as that simplifies design a lot.

Existing IPs can be connected to each other to wires on the board layout and new ones can be added using the Vivado IP Packer, described in the corresponding chapter.

After the design is implemented, one should simulate it as thoroughly as possible. The integrated simulation environment is capable of setting inputs to various levels and even simulating clocks. There are three stages at which Vivado supports simulations. As seen in the corresponding documentations [14][17][18] the first simulation is behavioral simulation, the highest level one, containing no specifics on how the design is implemented. It is the simulation which is the fastest to be completed, but is providing minimal information. Structural simulation is performed after synthesis and hence includes more information. Timing simulation considers the worst case placement and routing details of the design and hence takes the longest to be complete but identifies most of the problems.

As those potential problems are fixed in the board-layout or in VHDL Code and the design works as intended in simulation, one can head on to Synthesis. As seen [25], Synthesis is the automated process that translates the algorithmic description of the behavior and creates digital hardware that performs the described task.

Synthesis creates several modules performing different small tasks. Hence they need to be placed on specific parts of the desired chip and then be connected to one another. This step is called placement and routing or

implementation. One can define some physical and timing constraints here to make sure that, for example, the inputs of the digital design match the inputs on the board and that timing requirements are met.

As this, also automated process, is completed, one should analyze the results and can generate a bit-stream of the hardware, being a binary representation of the configuration data for the PL, which then can be programmed onto the chip.

Vivado has many other features, one of them being the capability of in-circuit debugging with Integrated Logic Analyzer (ILA) IP-Cores. The latter enable various debugging capabilities ranging from analyzing simple clocks up to full debug of AXI streams. After these debug operations are performed on the device, one can jump back to the board layout or even the VHDL-Code to improve on the design.

Another commonly used feature would be the Software Development Kit (SDK), used to generate custom Linux builds for targeted applications as well as bare metal applications.

In the so-called PetaLinux, one can embed hardware description, such that the device boots with the correct PL configuration. It also includes various pre-built drivers for existing IP-cores and supports advanced memory mapping and reservation. The latter is very important when using DMA, as described in 2.2.5.

2.4.2 Vivado IP Packager

Vivado IP Packager is used to package VHDL or Verilog code such that it can be processed as custom IP, which then can be used in the Board Layout with drag and drop. Custom IP supports various input/output ports ranging from simple 1 bit lines, up to AXIs. As the custom IP-Cores are treated just like separate projects, reconfiguration, simulation and debugging can be done easily.

As seen in the corresponding user guide [15] one can use IP Packager to combine VHDL source files, simulation model sets, example designs, test-

benches, documentation as well as entire block designs into one IP and hence modularize one's design.

2.4.3 Hardware Description Languages - VHDL

As previously discussed, there are many hardware description languages (HDLs). It shall be noted that functionality can be described with such a language than might be possible to synthesize or implement. As VHDL is used in this implementation, it will be explained more thoroughly.

One of the many guides on VHDL "Basic VHDL", by the RASSP Education & Facilitation Program [3] describes a single VHDL to consist of one entity and one or more architectures.

2.4.3.1 VHDL Entities

A VHDL entity, in the simplest form, contains a name and the description of the Port. The latter defines which interfaces connect the entity to the outside world. These can be either inputs, outputs or both. A short example of a clocked half-adder, having the inputs *a* and *b* is provided in Listing 2.13.

```
1 ENTITY half_adder IS
2     PORT ( a : IN STD_LOGIC;
3            b : IN STD_LOGIC;
4            clock : IN STD_LOGIC;
5            result : OUT BIT);
6 END half_adder
```

Listing 2.13: Half-Adder Entity

2.4.3.2 VHDL Architectures

A VHDL architecture provides one implementation of the VHDL entity. There may be multiple architectures, for example those utilized on different kinds of hardware, or simply one for simulation and one for implementation. A possible behavioral architecture for the previously defined half-adder is given in Listing 2.14.

```
1 ARCHITECTURE behavioral of half_adder IS
2     BEGIN
3         PROCESS (a,b,clock) BEGIN
4             IF rising_edge(clock) THEN
5                 result <= a AND b;
6             END IF;
7         END PROCESS;
8 END behavioral;
```

Listing 2.14: Half-Adder Timed Process

The most important parts start in Line 3, where the process begins. This process is equipped with a sensitivity list. This list contains all signals that shall trigger the process during simulation, when they are changed, which means that the process is triggered if one of the signals in the sensitivity list is changed. Furthermore the process first checks if there is a rising edge on the clock, then it performs the calculation.

If one does not care about the clocking of output, one can also use a data flow specification of the half-adder in Listing 2.15.

```
1 ARCHITECTURE behavioral of half_adder IS
2     BEGIN
3         result <= a AND b;
4 END behavioral;
```

Listing 2.15: Half-Adder dataflow representation

The main advantage of this implementation is resource usage. On the other hand it is, by itself, not a timed application, as the result will be completely independent from any clock.

2.4.4 Advanced eXtensible Interface

AXI is an interface standard developed by ARM® , used by many micro-controllers including the Zynq platform. As seen in the corresponding data-sheet by Xilinx [12] its main advantages are the consolidation of many interfaces into one and that the optimization on highest performance, maximum throughput as well as lowest latency can be done by interface specialists.

This interface is intended to connect various IPs to one another, as well as to manage the communication between PS, PL and memory. As this standard is broadly accepted in industry, it is used by other manufacturers as well.

2.4.5 Device Tree

The device tree is used by almost every linux based operating system to describe all devices part of a system. It contains the most basic components, such as the processor and memory, more specific ones such as DMA controllers. In the graphical example of a device tree in Figure 2.8, one can see its hierarchical structure. Every layer can have some children and various options, such as which clocks are connected and which memory address space is assigned to this device.

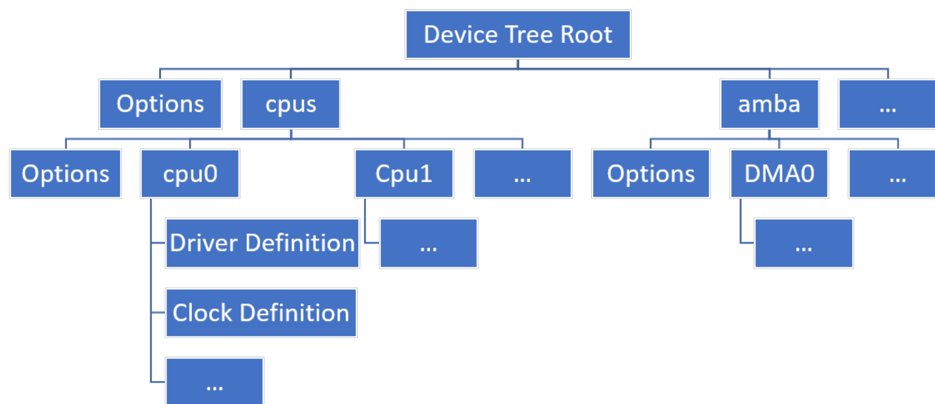

Figure 2.8: Device Tree graphical representation

2.4.5.1 DMA implementation

In the case of embedded development, especially with the Zynq platform, it is common to implement devices in the PL part of the chip and make the PS part aware of them via the device tree, where space in memory can be mapped for both to access, as seen in Section 2.5.2. This device tree is also used by the operating system to decide which drivers shall be

loaded. In Listing 2.16 the entry of a DMA core is given and will be further explained.

The first line must contain some arbitrary name. Henceforth one can define clock names in Line 3 and their respective implementations in line 4. The compatible part from Line 5 defines which driver shall be used to interact with this device. *reg* in line 6 denotes the base address and the memory width of the device. Furthermore there are extras, such as device and driver specifications in lines 7 and 8, options such as the option that scatter-gather engine is included.

Finally, a child of the device is defined in Line 10. In this case it is a DMA Channel performing the Memory to Stream (MM2S) operation, again denoting name, driver and device specific options.

```
1  dma@80410000 {
2      #dma-cells = <0x01>;
3      clock-names = "s_axi_lite_aclk\0m_axi_sg_aclk\0
       m_axi_mm2s_aclk";
4      clocks = <0x09 0x09 0x09>;
5      compatible = "xlnx,axi-dma-7.1\0xlnx,axi-dma-1.00.a";
6      reg = <0x80410000 0x10000>;
7      xlnx,include-sg;
8      xlnx,sg-length-width = <0x0e>;
9  
10     dma-channel@80410000 {
11         compatible = "xlnx,axi-dma-mm2s-channel";
12         dma-channels = <0x01>;
13         xlnx,datawidth = <0x10>;
14         xlnx,device-id = <0x02>;
15     };
16 };
```

Listing 2.16: Device tree DMA definition

2.4.5.2 Memory Reservation

The default setting for the operating system would be to use the whole memory region available for itself. As in most cases DMA is utilized as well, one needs to tell the operating system which part of the memory may be used and which is off-limits. This also happens via device tree.

One might suggest that it would be possible to acquire a big enough chunk of memory using *malloc*. When using this method, one cannot choose where the memory is located and it will almost certainly be partitioned and hence not be contiguous. These factors make transferring data from the software side to the hardware impossible.

In Listing 2.17 a memory region has been reserved for other devices and declared off-limits for the operating system in line 6. The user can access both regions, but the operating system will not put any of its own data there and will not automatically assign the regions to programs requesting memory. For each memory block there is an arbitrary name defined in the same line, a driver definition in line 7 , device specific options in lines 9 and 10 and the base address with the total range in Line 11.

```
reserved-memory {
    #address-cells = <0x01>;
    #size-cells = <0x01>;
    ranges;

    dma_desc@37E00000 {
        compatible = "shared-dma-pool";
        device_type = "reserved_memory_descr";
        no-map;
        linux,cma-default;
        reg = <0x37e00000 0x200000>;
    };
}
```

Listing 2.17: Device tree memory reservation

2.5 Server-Client Interfaces

Whenever it is necessary to split execution and control of a program onto different machines, Server-Client interfaces come to play. The server denotes the program or machine that is receiving instructions from the client and executing them. In general, the client would be more lightweight and very portable, meaning that it can be run on different machines. In the case of embedded applications the server needs to be lightweight as well, as it is deployed on a lower power system than the average PC, for example a

ZedBoard. It also helps to keep the server portable, as it can be deployed on other embedded systems.

2.5.1 QT Applications for ARM Architectures

According to Blanchette and Summerfield [2] QT is a software framework and combined with QT Creator, an IDE for various programming languages, one of the most commonly used being C++ and QT, it provides the designer with the possibility of one codebase with portability to many different operating systems. In the case of embedded development it offers a wide set of libraries and QT-specific code snippets to develop applications. Like many IDEs, Qt Creator allows many different automation tasks, such as enabling automatic deployment of the embedded application onto the target device. Hence the application can be developed on any system and then cross-compiled for the target architecture and run and debugged remotely from the development system.

2.5.2 Memory Mapping and Interfacing

For various reasons, for example to access special function registers directly, it might be necessary to reserve a certain address space in memory for special use and prevent the rest of the system from messing with the stored data.

The obvious choice when mapping memory would be to use the library *sys/mmap* with its function *mmap*. When the user requests a memory region, using the provided function *mmap*, the operating system tries to allocate the requested request. The address argument provided is a hint to the kernel which area of memory it should chose. It will always pick a nearby page boundary which is above or equal to the address value specified. This is not practical when one wants to transfer data from one processing system onto the PL or vice versa, as the PL does not care about page boundaries and transfers the exact address.

Instead one should take advantage of the fact that linux treats every device as a file. If sufficient access rights are available, one can map and henceforth access the linux file */dev/mem* corresponding to the whole of the system memory and writing to and reading from it. The QT Library QFile, which allows very easy mapping of specific regions of a file and thus treats the mapped file as the desired memory region that can be used here.

2.5.3 QT Applications for Desktop Architectures

As previously mentioned QT is very well capable of using cross-compilers to develop applications for various architectures. Another big advantage of using QT is its capability of designing user applications and their GUI using various libraries, pre-built functions and buttons using drag and drop. One can for example drag a new button onto the QT GUI using the designer view and then edit its function using QTs libraries or falling back to C++ code.

2.5.4 Communication Standard

Data used within programs has a data type, *int* for example. This works fine within the program, but as soon as the data needs to be transferred over to another machine, the datatype needs an additional definition to make sure that both machines interpret it the same way. Hence one needs to define and maintain a communication standard. The existing standard for this application is based on the ethernet interface, serializing or converting the data to a JavaScript Object Notation (JSON) string, transferring this byte string and then deserializing or restoring it again.

Using this method one can define commands in the server and the client application, one of which would be sending a command, the other receiving and acting on it.

3 Implementation

This chapter focuses on the specific implementation of the given project assignment. First there will be a short overview on the used hardware and its components, then there will be an elaboration on the automated tasks regarding embedded development.

Additionally, the logic circuits used within this project will be explained and finally it will be shown how the functionality is controlled on the part of the server and client.

A graphical representation of the individual parts of the Project, including existing ones in green is shown in Figure 3.1.

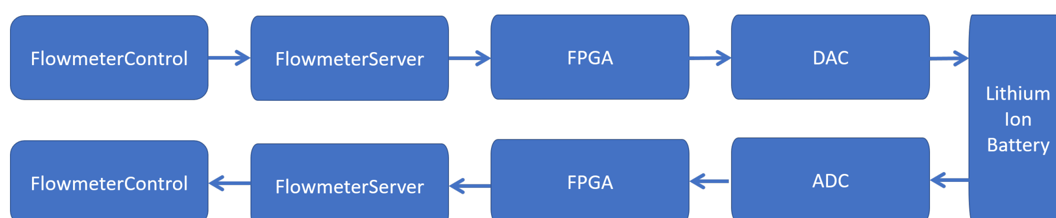

Figure 3.1: Block diagram of a measurement flow

3.1 Hardware

The hardware used consists of a PicoZed, sold by Avnet Inc. [11] board as well as a custom board, developed by Reinhard Klambauer and Alexander Bergmann [20].

3.1.1 Xilinx Zynq®-7000 System-on-Chip

As seen in the product description [11], the PicoZed is a SoM, based on a Xilinx Zynq All Programmable System-on-Chip. Over 100 I/O pins are accessible via the three I/O connectors at the bottom of the module, providing maximum flexibility for implementation.

3.1.2 Carrier Board

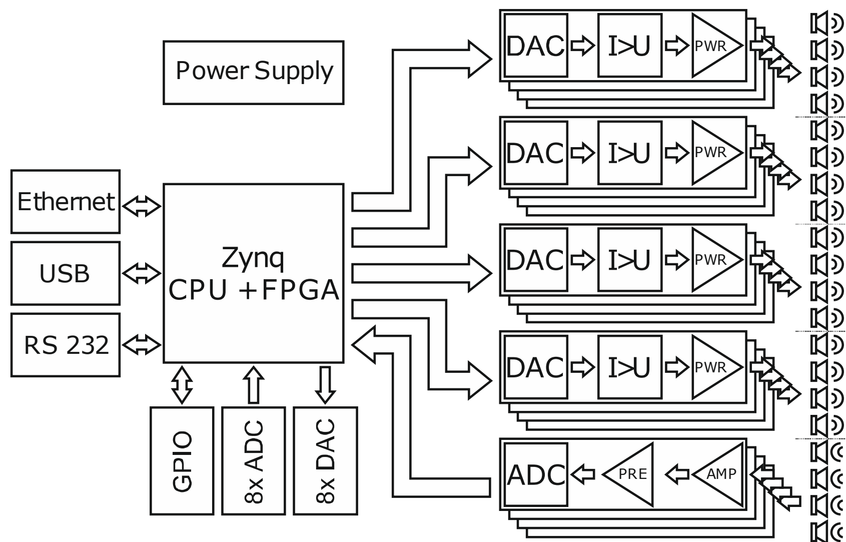

Figure 3.2: Schematic of the Zynq board with paths for ultrasonic signal generation and sensing by R. Klambauer, A. Bergmann [20]

The carrier board used, developed by Reinhard Klambauer and Alexander Bergmann, [20], which is depicted in Figure 3.2, provides access via required interfaces. It includes for example a gigabit ethernet connector, a serial interface connector as well as a Joint Test Action Group (JTAG) one. The former is used for normal operation, when communicating with the control application using the communication standard, described in Section 3.4.3.

The serial connector is mainly used in fault conditions, as the boot-up of the PS or PetaLinux prints its debug log there. Command-Line commands

can also be sent to the operating system via that same interface. This comes in especially handy when dealing with custom PL modules and IP-cores, as the drivers might not always work out of the box and some debugging might be necessary.

The JTAG connection is used for development purposes, as Vivado allows the use of a Platform Cable Usb II, the functionality of which is described in the respective data-sheet [16]. In short, it allows the easy transfer of new hardware definitions for the PL part of the chips onto itself. Apart from programming, the use of debug options an utilizing integrated logic analyzers simplify development.

In addition to the stated interface connections, the carrier board does also provide multiple DACs, ADCs and their respective necessary components for operation and voltage level conversion. As seen in the respective data-sheet [10], each of the 16 DACs AD9707 supports up to 175 MSPS update rate and a 14 bit resolution. They can be individually controlled, via four separate outputs of the PL, each of which utilized time domain multiplexing, meaning that there is a phase difference of 90 degrees in between the individual 30 MHz clock signals, to control four different DACs. To achieve this, the data-path needs to be clocked with 120 MHz.

3.2 Packaging

3.2.1 Shell Scripting and Automation

The most strenuous tasks performed within the setup and build process have been documented and automated. The respective automations will be discussed here.

3.2.1.1 Installation of QT for cross-compiling for ARM®

To obtain an executable capable of compiling *QT*-code into applications for a target architecture that is not default, one needs to download, modify and build *QT* oneself. This process has been widely discussed in community

forums, as most people catch problems when trying to do as described in the provided documentation [6] [24]. The process is automated with Listing 3.1 and described below.

```
1  WORKING_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" > /dev/
       null 2>&1 && pwd )"
2  source $WORKING_DIR/../config
3
4  mkdir $QT_INSTALL_DIR
5  cd $QT_INSTALL_DIR
6  rm -R install
7  mkdir install
8  rm -R build
9
10 ZYNQ_QT_BUILD=$PWD/build
11 ZYNQ_QT_INSTALL=$PWD/install
12
13 rm *.tar.gz*
14 wget $QT_LINK
15 tar -xvf $QT_VERSION.tar.xz
16 mv $QT_VERSION build
17
18 PATH=$VIVADO_PATH/$VIVADO_VER/gnu/aarch32 /lin/gcc-arm-linux-
       gnueabi/bin/:$PATH >> /etc/bash.bashrc
19 PATH=$VIVADO_PATH/$VIVADO_VER/gnu/aarch32/lin/gcc-arm-linux-
       gnueabi/bin/:$PATH
20
21 sed -i 's/arm-linux-gnueabi-/arm-linux-gnueabihf-/g'build/
       qtbase/mkspecs/linux-arm-gnueabi-g++/ qmake.conf
22
23 echo export PATH=$VIVADO_PATH/$VIVADO_VER/bin:$PATH >> /etc/
       bash.bashrc
24 PATH=$VIVADO_PATH/$VIVADO_VER/bin:$PATH
25
26 echo export PATH=$ZYNQ_QT_INSTALL/bin:$PATH >> /etc/bash.bashrc
27 PATH=$ZYNQ_QT_INSTALL/bin:$PATH
28
29 cd build
30 source /etc/bash.bashrc
31
32 SK\gls{ip}_PACKAGES=`(find -maxdepth 1 -type d  grep qt
        sed 's/.\/qt/-skip /g'  sed 's/-skip base//g'  tr '\n' ' ')`
33 #Skip additional base packages
```

```
34 NO_PACKAGES="-no-gif -no-libjpeg -no-mtdev -no-sql-db2 -no-sql-
      ibase -no-sql-mysql -no-xcb -qt-freetype -no-fontconfig -no-
      harfbuzz -no-xcb-xlib -no-cups -no-iconv -no-icu -no-eglfs -
      no-openssl -no-opengl"
35
36 ./configure -xplatform linux-arm-gnueabi-g++ $SKIP_PACKAGES
      $NO_PACKAGES -opensource -confirm-license  -prefix
      $ZYNQ_QT_INSTALL
37 echo "Configure Complete"
38
39 make -j 2
40 echo "Make complete"
41
42 make install
```

Listing 3.1: Installation script for QT for PetaLinux

First, one needs to prepare an installation directory, defined in the separate config file. After this has been done, any old remainder of QT sources are removed and the current source is fetched from Line 13 onwards.

Lines 18 and 19 make sure that the *GCC* compiler suitable for the architecture is reachable from shell, by adding it to the local and global PATH variable.

As there is no base config for the desired architecture, one needs to edit an existing similar one, to match the correct GNU Compiler Collection (GCC) version in Line 21.

From Line 23 the script makes sure that the Vivado and QT binary directories are reachable from command line, by adding them to the PATH variable.

To make sure that the whole PATH is available within the script, the bash config is sourced in Line 30.

When building QT, there are various additional modules included by default. They assist in developing applications, but including them in the build process significantly increases build time. As no additional packages are necessary for this application from Line 32, it generates a list of packages to skip from the packages available. Some packages which are not caught by this process are hardcoded in Line 34.

Lastly the configuration process is started in Line 36 as well as is the build process in Line 39. When the timely operation of building of such a huge application is finished, it is installed in Line 42.

3.2.1.2 Cable Drivers for Platform Cable

This short script in Listing 3.2 describes and performs the installation of the drivers necessary to use the functionality described in Section 3.1.2. After reading the global configuration file it runs the installation script from Xilinx.

```
1 #Source configs
2 WORKING_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" > /dev/
    null 2>&1 && pwd )"
3 source $WORKING_DIR/../config
4
5 # Path to Xilinx SDK Cross-compiler
6 cd $VIVADO_PATH/$VIVADO_VER/data/xicom/cable_drivers/lin64/
    install_script/install_drivers/
7 ./install_drivers
```

Listing 3.2: Installation script for Cable Drivers

3.2.2 Complementary Scripts

3.2.2.1 Matlab Script to generate an output window

The script presented in Listing 3.3 is used to generate one period of any signal in the correct format to be played on the discussed Flowmeter.

In the first relevant Line number 4 the output frequency is defined. The clock for the DAC providing the output is fixed at 30 MHz, hence the desired frequency is proportional to the necessary amount of values provided to the window generator. As the amount of values is calculated, an array ranging from 1 to the amount of values is generated in line 11.

The ADC has 14 bit, hence the maximum value provided to it is in the range -8192 to 8191. To keep the matlab script simple the range is considered to be -8191 to 8191 by setting *data_width* to 2×10^{14}-2, to avoid rounding overflows. In lines 17 to 21 the basic implementations of sinusoidal, cosine windows and sawtooth signals are shown. A combination of two simple signals is depicted in Line 23.

At last the generated signal is displayed using a figure and exported to a CSV-file that can be imported into the FlowmeterControl Application.

```
close all

% config tested from 14600 to 300000
frequency = 14600;

% calculation of time scale values
period = 1 / frequency;
f_clk = 30 * 10^6;
time_per_tick = 1/ f_clk;
number_of_values = period / time_per_tick;
N = 0: 1 : number_of_values;

% calculation of values
adc_depth = 2^14-2;

%sine
F1 = adc_depth/2* sin(N/number_of_values*2*pi);
%cosine window
F2 = adc_depth/2 * sin(pi * N / number_of_values);
%sawtooth
F3 = int16(adc_depth/2-adc_depth/2*2*N/number_of_values);

F = int16(F1.*(F2./adc_depth)/32);

% plot and export values
figure
plot (N,F,'.')
csvwrite('window_values.csv', F)
```

Listing 3.3: Script for generating new window data

3.2.2.2 Other Scripts

Some other smaller scripts have been implemented to fix machine related network problems, screen resolution adjustments on virtual machines as well as connection scripts for ssh and serial access to PetaLinux. As they are well commented and consist only of simple code, they will not be evaluated further.

3.3 Embedded Software - Xilinx

In this Section the implementation of the desired hardware, mainly performed in Vivado and the corresponding SDK tools is described. The preexisting configuration is described by Florijan Reichmann [31] and will not be elaborated further.

3.3.1 Device Tree

The basics of device trees and also the necessary components for this project have been described in Section 2.4.5. In addition to the definitions of the components, which at best are done by *petalinux-build* and *petalinux-package*, described in Section 3.3.5, one should always check the correct implementation and in some cases edit it to match new requirements missed by the automatic configuration.

The device tree generated by the previously mentioned build tools is given in Device Tree Blob (DTB) format, which is perfectly machine-readable, but not human-readable. To generate the Device Tree Source (DTS) file one can use the executable dtc to convert in between the two formats as described in the corresponding reference [33]. Line number 1 is used to generate a source file from the blob, while Line number 2 does the opposite.

```
./dtc -O dts -o ../dtc.files/system.dts ../dtc.files/system.dtb
./dtc -O dtb -o ../dtc.files/system.dtb ../dtc.files/system.dts
```

Listing 3.4: Commands for Device Tree compilation and decompilation

3.3.2 Address Map Layout

The chip used is equipped with a total of 1 GB of System memory, but like in most systems, the addressable range is greater than system memory, due to other devices being addressable as well.

The most important address ranges in this application have been taken from [9] and depicted in Table 3.1. As one can tell from the first line, there is one

GB of memory, most of which is usable and there are two more GB, used for memory mapped transfer of data via AXI Streams, as described in 2.4.4. The two DMA IP-cores responsible for custom signal generation are using the second AXI channel, while all other cores are using the first one.

Start Address	**Stop Address**	**Purpose**
0x0000 0000	0x3FFF FFFF	System Memory
0x4000 0000	0x7FFF FFFF	M_AXI_GP0
0x8000 0000	0xBFFF FFFF	M_AXI_GP1

Table 3.1: Memory Layout

All the devices using these address spaces can be seen in Table 3.2. The devices *DMA0*, Signal Generator and ADC Sampler have been implemented by the preceding projects and are only added for completeness. For the functionality of the individual devices, refer to Section 3.3.3.

3.3.3 Vivado IPs

Various existing IPs have been used in addition to the custom ones described in Section 3.3.4. Their use and configuration are described in this chapter.

As seen in Figure 3.3 the PS is used to provide the *AXI GPIO* and *AXI Direct Memory Access* IPs with data. They are connected using *AXI Interconnects*. The *AXI Direct Memory Access* IP reads data from memory and pushes it onto the *AXI4-Stream Data FIFO* which stores all of it at once and releases 2 bytes at a time to the *Delay Generator*. After processing the values from the 32 bit *AXI GPIO* is done, it passes the data onto the *Output Switch*, which is controlled by the 1 bit *AXI GPIO*. The *Output Switch* passes the data onto the DACs.

3.3.3.1 AXI Direct Memory Access

As seen in the Product Guide on AXI DMA [13], the zynq platform used, an IP-core acting as a DMA engine is provided by Xilinx. Each added core provides read and write access to memory on possibly multiple channels. As

Start Address	Stop Address	Usage
0x33E0 0000	0x33E7 FFFF	DMA 1 Descriptors
0x33E8 0000	0x33EF FFFF	DMA 2 Descriptors
0x3400 0000	0x341F FFFF	DMA 1 Data
0x3420 0000	0x343F FFFF	DMA 2 Data
0x37E0 0000	0x37FF FFFF	DMA 0 Descriptors
0x3800 0000	0x3FFF FFFF	DMA 0 Data
0x4100 0000	0x4100 FFFF	DMA 0
0x4120 0000	0x4120 0FFF	GPIO Delay 0
0x4121 0000	0x4121 0FFF	GPIO Delay 1
0x4122 0000	0x4122 0FFF	GPIO Delay 2
0x4123 0000	0x4123 0FFF	GPIO Delay 3
0x4124 0000	0x4124 0FFF	GPIO Delay 0
0x4125 0000	0x4125 0FFF	GPIO Delay 1
0x4126 0000	0x4126 0FFF	GPIO Delay 2
0x4127 0000	0x4127 0FFF	GPIO Delay 3
0x43C0 0000	0x43C0 0FFF	Signal Generator
0x4400 0000	0x4400 FFFF	ADC Sampler
0x8040 0000	0x8040 FFFF	DMA 1
0x8041 0000	0x8041 FFFF	DMA 2

Table 3.2: Device Address Layout

one core for multiple channels is not recommended by Xilinx, it is advised to be using one IP-core for each channel.

One can define if the core should provide read or write functionality respectively. The other settings used define the size of the transferred data. First, there is the Memory Map Data Width, describing how big one data chunk is in memory. This can also be seen as a memory partition or how much data is described per descriptor. This has of course got to be consistent with data width when transferring the data to memory from the software side. Additionally, Stream Data Width is set, defining the size of the output data. As it would be very inefficient to transfer only one memory data chunk at a time, one can set a burst size, defining the amount of data chunks transferred every time the memory is accessed. The core includes a data

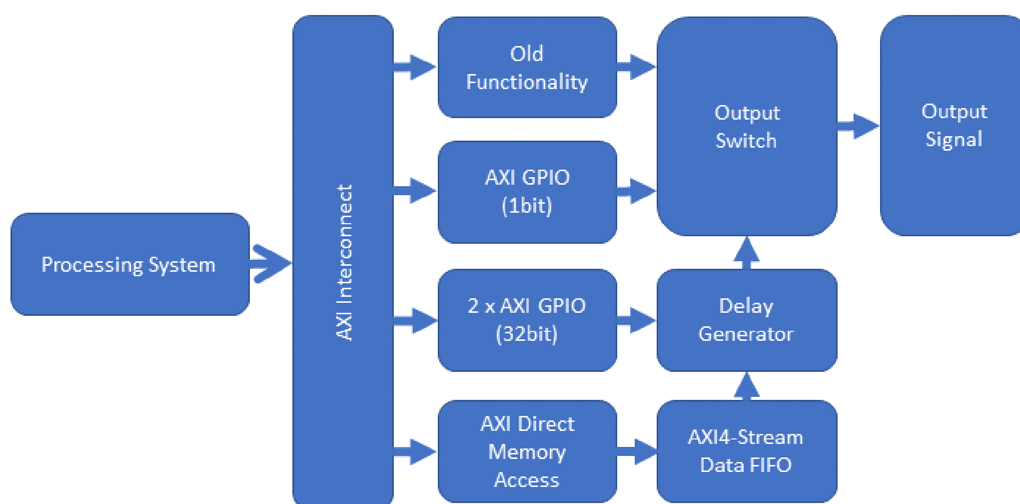

Figure 3.3: Block diagram hardware

buffer to store bytes to be transferred.

In general, there are two operating modes for this IP-core, the first one is Direct Register Mode. This is the more basic functionality where the user does not need to care about memory addressing of the individual chunks of data. As the core is started by setting the run bit and a valid source address is provided to the *MM2S_SA* address, it transfers the data. The main disadvantage is that one single and continuous address space in memory has to be provided for all transmitted data and the fact that when the end of the address space is reached, the transfer stops. Hence one cannot reliably provide an address space containing, for example, one period of a signal and transferring that signal to a DAC continuously.

For more complex applications the second mode, Scatter Gather Mode comes to hand. As touched on before, it provides the advantage of continuous or cyclic transfers. In this mode the user has to provide the scatter gather descriptors in memory. One such descriptor basically contains the address of the data and the address of the descriptor. The DMA engine is provided with an address of the first and the last descriptor. The first or starting descriptor needs to be written to the *Current Descriptor* register. One can

now enable interrupts and set the run bit in the control register. As the last or *Tail Descriptor* address is set, the transfer begins from source to tail. It will in any case pause as the last descriptor is reached. To achieve cyclic transfer one can set the address of the next descriptor of the last descriptor from the cyclic chain to the first descriptor. Additionally, one shall set the address of the last descriptor for the DMA to some address that can never be reached. Henceforth the DMA will continuously transfer the memory described by the descriptor chain.

This DMA engine is capable of reading from memory and writing to memory. The other end of the data is always an AXI interface. Those two directions are called Stream to Memory (S2MM) and MM2S, MM denoting memory and S denoting the AXI Stream. The first noted is always the source, the second is the destination of the data flow.

The enabled Scatter Gather Engine is controlling the MM2S channel with a data width of 128 bit. By setting the output width to 16 bit, the 14 bit DAC can be fed every clock cycle. Memory data width has been determined empirically. In general, you need to balance all implemented configuration options such that the buffer is large enough, that it provides new values until the other DMAs finish their transfer. On top of that one needs to see to it that the transfer does not take longer than the buffers of the rest of the DMAs can hold.

At first four DMAs for four different signal windows were implemented, but it became obvious that this would not work, because there is only one memory channel and while one of the cores is transferring data, the others cannot. Unfortunately, when using more than two DMA cores, the transfer is not fast enough to provide consistent output signals.

3.3.3.2 AXI4-Stream Data FIFO

A First in First Out (FIFO) core is basically a buffer taking bursts of data at once and outputting them one per clock cycle. This acts as a second buffer, as the buffer included in the DMA core is for the memory mapped side of the transfer.

Regarding settings of this IP-core, one can set the FIFO-depth or the size of the buffer as well as define a custom output data width, if needed.

3.3.3.3 AXI Interconnect

AXI Interconnects are IP-cores provided by Xilinx Inc., which manage the connection of multiple slave interfaces to one master interface. As the PL controller provides only two general purpose AXI Masters, these interconnects were used to control the numerous other components.

3.3.3.4 AXI GPIO

The General Purpose Input/Output (GPIO) IP-cores provide very basic, but also very easy to implement data transfers from the PS side of the system to the PL part and vice versa.

They are on the one side connected to the AXI Interface, and on the other side provide up to two individual inputs and outputs with up to 32 bit each, as described in the respective product guide [7].

In this project four of them have been used on the one hand to define the amount of ticks to play the signal and pause the signal on the four different outputs. Each of them only has outputs with 32 bit enabled.

On the other hand, another four of them have been used to switch between the old functionality of providing only sinusoidal output signals and the new functionality of having custom windows defining the signal. In this case only one 1 bit output per IP-core is enabled.

3.3.4 VHDL Implementation

This Section intends to describe developed IP-cores for this project. In addition to the pre-existent cores two more have been implemented in various design steps. The final version of the cores will be the only elaborated one.

3.3.4.1 Delay Generator

The first and main IP combines most of the new functionality on the PL side of the system. The individual ports of this core are depicted in Table 3.3.

The first two elements, *clk* and *reset* are the input clock as well as the reset signal, providing timed operation and setting all data back to the initial values, respectively.

The next element *play_ticks* precisely defines how many clock cycles the input signal *signal_in* should be played onto the output port *signal_out*. The similarly named *pause_ticks* defines the minimum amount of clock cycles to wait after one period has finished to start a new one. The real number of clock cycles waited will, in most cases, be higher than the requested one, as the memory mapped window is sent to the input of this core continuously and it has to be made sure that the starting point of the output signal is also the first value of the desired signal.

This has been achieved by making sure that the desired signal can never be at value 1 , and sending the value 1 always as the first value, signalizing this IP-core that a new signal period on the input has started and it can now start generating an output, of course only when pause time has exceeded.

Port Name	Direction	Data Type	Size (bit)
reset	in	STD_LOGIC	1
clk	in	STD_LOGIC	1
play_ticks	in	STD_LOGIC_VECTOR	32
pause_ticks	in	STD_LOGIC_VECTOR	32
signal_out	out	STD_LOGIC_VECTOR	16
signal_in	in	STD_LOGIC_VECTOR	16

Table 3.3: Ports in Delay Generator

Additionally, to the input signals, there are also internal signals for synchronization and data storage, as described in Table 3.4. The first signal *read_pointer* is intended to make sure that the four individual DACs controlled by this core receive the correct values. It is increased every clock cycle and determines which channel is executed in every iteration of the program.

Each of the channels has a signal storing how many clock cycles in the current state, being either play or pause, have been executed. When the program is currently providing an output signal, the *play_ch* counter of the corresponding channel is increased, if it is not providing an output signal, the *pause_ch* counter is increased.

Signal Name	Data Type	Size (bit)
read_pointer	integer	0 to 3
play_ch1	STD_LOGIC_VECTOR	32
play_ch2	STD_LOGIC_VECTOR	32
play_ch3	STD_LOGIC_VECTOR	32
play_ch4	STD_LOGIC_VECTOR	32
pause_ch1	STD_LOGIC_VECTOR	32
pause_ch2	STD_LOGIC_VECTOR	32
pause_ch3	STD_LOGIC_VECTOR	32
pause_ch4	STD_LOGIC_VECTOR	32

Table 3.4: Signals in the Delay Generator

The VHDL code providing the said main functionality for channel 1, is provided in Listing 3.5. The other channels, as well as the reset functionality and *read_pointer* increase have been removed for better readability.

At first it is determined whether the current state is tasked to generate a signal or not with an if-else-clause in lines 1 and 5. Next the counter for the respective channel and state is increased in lines 2 and 6.

While generating an output in Line 3 the pause counter is reset in Line 4. Whilst not generating an output, 0 is written to *signal_out* in Line 7 and another if-clause in Line 8 checks if the pause is theoretically completed as well as if the input is currently providing the start of a period. If that was the case, the play counter is reset in Line 9, to jump back to the playing state.

```
1 if (play_ch1 < play_ticks ) then
2     play_ch1 <= std_logic_vector(to_unsigned(to_integer(
    unsigned(play_ch1 )) + 1, 32));
3     signal_out <= signal_in;
4     pause_ch1 <= X"00000000";
```

```
5  else
6      pause_ch1 <= std_logic_vector(to_unsigned(to_integer(
       unsigned(pause_ch1 )) + 1, 32));
7      signal_out <= X"0000";
8      if (pause_ch1 > pause_ticks and signal_in = X"0001") then
9          play_ch1 <= X"00000000";
10     end if;
11 end if;
```

Listing 3.5: VHDL Code for Delay Generator Channel 1

3.3.4.2 Output Switch

This very simple IP is tasked to switch between the previous and the current functionality. The individual ports of this IP-core are described in Table 3.5.

As the previous functionality has not been touched, its output is connected to one of the signal inputs, while the output of the new functionality is connected to the other one. The other input *send_signal_1* can be seen as a boolean variable being either true or false. For backwards compatibility, the old function is enabled per default and the output of this IP directly connected to the DACs input. The VHDL process providing this functionality can be seen in Listing 3.6

Port Name	Direction	Data Type	Size (bit)
signal_1	in	STD_LOGIC_VECTOR	16
signal_2	in	STD_LOGIC_VECTOR	16
send_signal_1	in	STD_LOGIC	1
signal_out	out	STD_LOGIC_VECTOR	16

Table 3.5: Ports on the output switch

```
1 process (signal_1, signal_2, send_signal_1) begin
2     if (send_signal_1 = '1') then
3         signal_out <= signal_1;
4     else
5         signal_out <= signal_2;
6     end if;
7 end process;
```

Listing 3.6: Process for Output Switch

3.3.5 Updating PetaLinux

The custom operating system PetaLinux, was provided by Florijan Reichmann [31]. As there is no need to update the operating system in general, it was left as is and it was only the embedded hardware definition that has been updated.

Within Vivado it is possible to export a Hardware Description File (HDF) as well as the bit-stream of the hardware. Both of them are needed to generate the necessary PetaLinux components.

Xilinx Inc. published separate software incorporated in the PetaLinux installer to easily perform the required tasks. As one has access to the board support package, one can generate a project folder. After sourcing the PetaLinux tools using Line 1 from Listing 3.7, the project folder from [31] has been provided with a new hardware configuration using line 2. Now the build process can be started with Line 3 and when it completes the hardware it can be packaged into the correct format for the desired chip using the code from Line 4.

```
1 source /opt/petalinux2018_3/settings.sh
2 petalinux-config --get-hw-description $PATH_TO_HDF_FOLDER$
3 petalinux-build
4 petalinux-package --boot --u-boot images/linux/u-boot.elf --
    fsbl images/linux/zynq_fsbl.elf --force --fpga
    $PATH_TO_HW_BIT_FILE$
```

Listing 3.7: Important PetaLinux commands

As thoroughly explained in the PetaLinux command reference [8] this process generates 2 important files *system.dtb* and *BOOT.BIN*. The former is the automatically generated device tree blob, described in Section 3.3.1, the second file holds the boot image for PetaLinux, being the first and second stage bootloader as well as the hardware definition loaded onto PL on startup. This file needs to be written to flash memory of the Zynq chip, using for example the *flashcp* command.

3.4 Server-Client Interfaces

The project consists of the two applications FlowmeterServer and FlowmeterControl. The first is a C++ based application running on the Zedboard's PetaLinux, the second one is also a C++ application running on an Ubuntu Virtual Machine (VM). Both of them have been developed using QT Creator. In a production environment the both of them would be connected using an ethernet interface. This section describes the individual functions that make up the FlowmeterServer, FlowmeterControl as well as their communication.

3.4.1 FlowmeterControl Application

In this application graphical representations of the desired settings have been implemented. Those settings comprise the setting of the amount of ticks to play and pause the signal for four different channels, as well as the possibility to parse a *CSV* file to import a new window as can be seen in Figure 3.4.

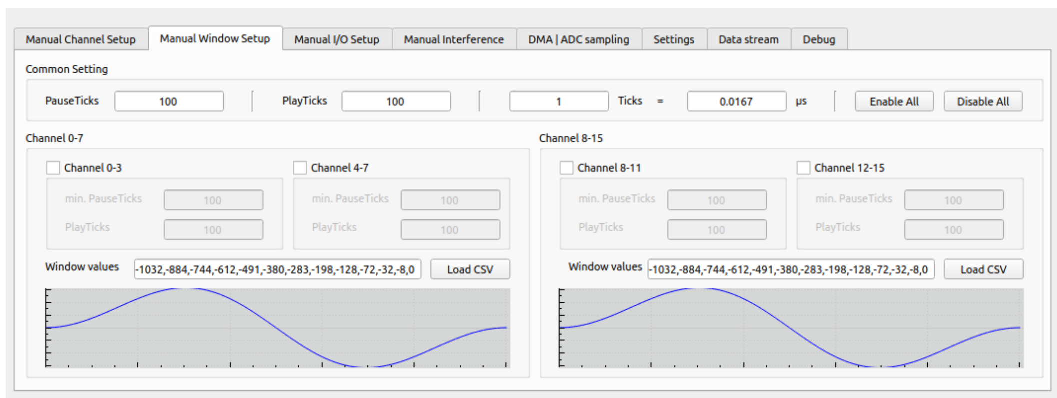

Figure 3.4: GUI representation in FlowmeterControl

Additionally, the Flowmeter is now feeding all outputs with the default sinusoidal signal of 300 kHz, when it is started.

3.4.2 FlowmeterServer Application

The FlowmeterServer application is receiving commands via the ethernet interface and acting on them. Hence it processes the data and sets the FPGA accordingly. The most important functions achieving DMA operation, as described in [13] are discussed here. A graphical representation of the data flow after a command is received by FlowmeterControl can be seen in Figure 3.5.

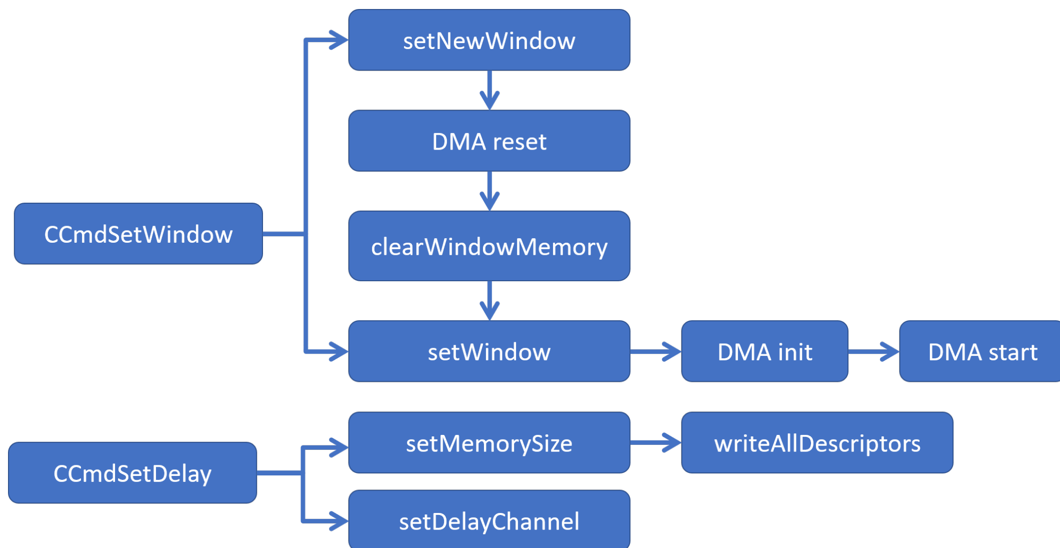

Figure 3.5: Block representation of the software functions after a command is received on the server

3.4.2.1 DMAHandler::setNewWindow

This function is intended to be called whenever a new window is desired. A window, in general, is the template for an output signal. It represents one period of the output which is played over and over again.

As arguments the function accepts a channel number and an initial value, which would be the first value of the new window. At first it resets the corresponding DMA (Section 3.4.2.5), to ensure any previous settings removed

and that it is stopped. Secondly, the *clearWindowMemory* (Section 3.4.2.2) and the *setWindow* (Section 3.4.2.3) functions are called. At last the corresponding DMA is initialized (Section 3.4.2.6) and started (Section 3.4.2.7)

3.4.2.2 DMAHandler::clearWindowMemory

This function, taking the channel number as an argument, is erasing the whole memory used by any potential previous window by writing 0 to it and additionally setting the first value to 1, to indicate to the Delay Generator (Section 3.3.4.1) that a new window is starting.

3.4.2.3 DMAHandler::setWindow

This high-level abstraction of a write operation to memory takes the channel number and a value as an argument and writes this data to the next memory address of the window. Additionally, it checks if the window exceeds the maximum allowed length of 2 MB. This window length allows for 262 144 values to be stored per channel. This can be calculated by dividing 2 MB by the 4 theoretical channels and the 2 bytes per value.

3.4.2.4 DMAHandler::setMemorySize

This function is intended to process the channel number and the amount of ticks to be played, or simply speaking the length of the window and adjusting the memory size for the DMA accordingly. This is done by first resetting (Section 3.4.2.5) the DMA, then initializing it (Section 3.4.2.6) and writing a new set of descriptors (Section 3.4.2.8) and finally starting the DMA again (Section 3.4.2.7).

3.4.2.5 DMA_WINDOW::reset

This functions performs a system reset on the DMA by setting the according bit in the control register and then waiting for a successful reset, or a timeout done.

3.4.2.6 DMA_WINDOW::init

This low-level function is performing the necessary tasks to initialize the DMA, consisting of enabling basic interrupts (on complete and on error) in the control register and then reading the status register to check if the reset was successful.

3.4.2.7 DMA_WINDOW::start

This function performs the main task of setting the correct DMA bits to start a memory transfer. At first it writes the current descriptor to the corresponding control register to tell the DMA where to start the transfer. Additionally, it enables the DMA by setting the Run/Stop bit and cyclic operation in general. The latter operation is used to ensure a continuous stream of window data. At last it writes the tail descriptor to the corresponding register. This is intentionally set to a register that is never reached, such that the transfer is never completed automatically, unless stopped intentionally.

3.4.2.8 DMA_WINDOW::writeAllDescriptors

This function is generating a descriptor chain, which is a series of single descriptors. Each Descriptor occupies 64 bit and consists of the address of the next descriptor, a memory location containing window data, the amount of bytes transferred out of the MM2S Stream as well as other configuration options, which are not used in this project.

After completing the chain it is made sure that the last descriptor has the parameter *next descriptor address* set to the first descriptor of the chain for continuous transfer.

3.4.2.9 WINDOW::init

This function first opens the device */dev/mem* corresponding to system memory and then mapping the eight GPIO IP-cores memory areas to the program. The first four GPIOs are used to set the playing and pausing amount of ticks for the respective delay generators and the remaining are enabling or disabling custom window generation.

3.4.2.10 WINDOW::setDelayChannel

This function utilizes the channel number, the ticks to play as well as the ticks to pause and if the custom window generation is enabled as parameters and further sets the corresponding values in memory at the previously mapped GPIO cores to pass it the programmable logic.

3.4.3 Communication Standard

To enable communication between the Server and the Control application the following commands have been implemented on both endpoints.

3.4.3.1 CCmdSetDelay

This class is considering the channel number, the amount of ticks to play and pause as well as if the channel is enabled. The first three are of the datatype *int*, the latter is a *bool*. They are serialized on the FlowmeterControl application, then parsed and deserialized on the FlowmeterServer. It is then calling the *setDelayChannel* (Section 3.4.2.10) and the *setMemorySize* (Section 3.4.2.4) functions.

3.4.3.2 CCmdSetWindow

This class is receiving the window values for the respective channel on the Control application, serializing them to parse them onto the Server. After they are deserialized there and then calling either the *setNewWindow* (Section 3.4.2.1) or the *setWindow* (Section 3.4.2.3) function, depending on if the first values or subsequent ones are sent.

4 Results

It can be said that it works to generate signals other than just the ones by Xilinx Inc. who provided sinusoidal ones. One period of the resulting signal with the settings visible in Figure 4.1 can be seen in Figure 4.3. One might think that this represents a sinusoidal signal, but one is mistaken as this is a sinusoidal signal multiplied by a cosine window represented mathematically by Section 3.3.

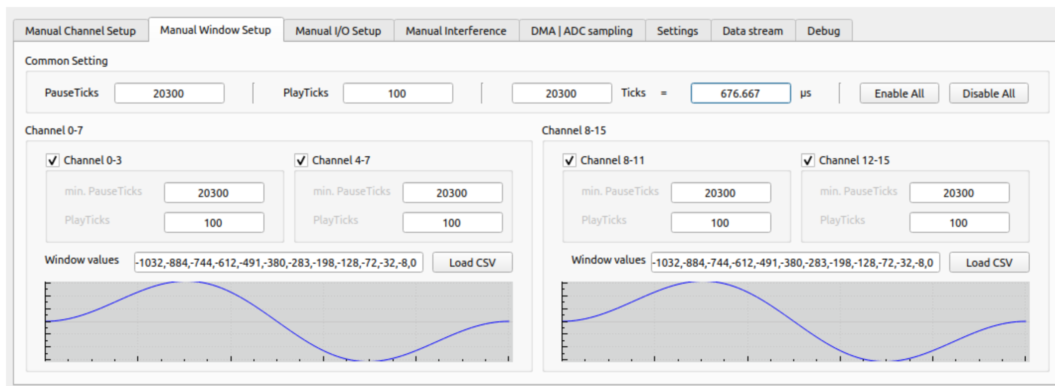

Figure 4.1: Settings for test signal

To showcase how well two different signals work together, the signal on channels 0 to 7 have been kept the same as in the last example and the other channels have been fed with one period of a sawtooth signal, as $F3$ in Section 3.3. The settings can be seen in Figure 4.4, the outputs in Figure 4.5.

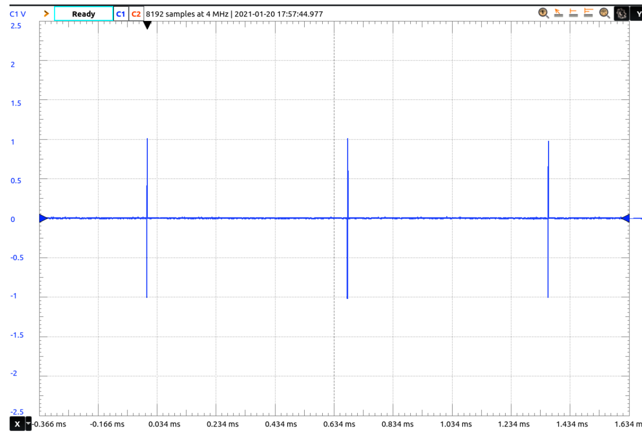

Figure 4.2: Resulting signal (multiple pulses)

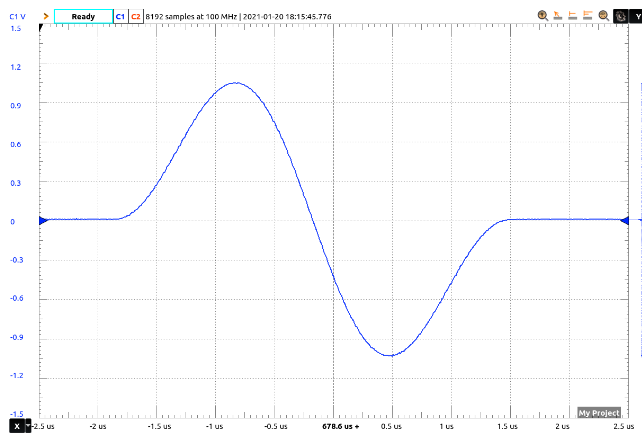

Figure 4.3: One pulse of the resulting signal

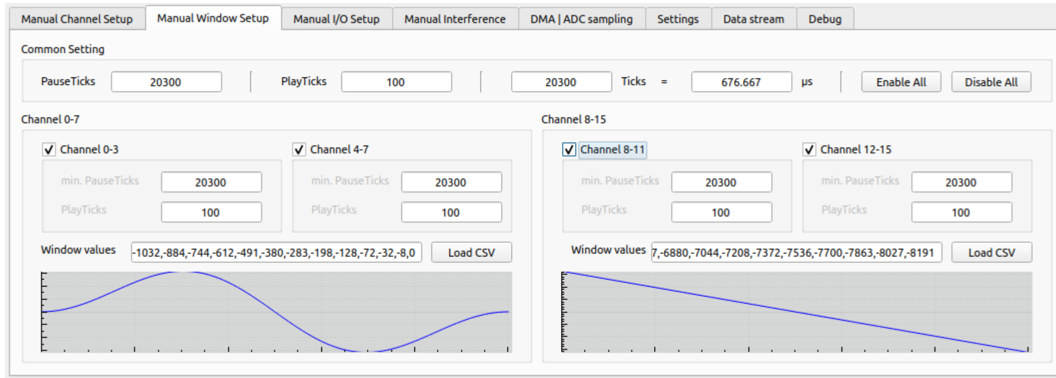

Figure 4.4: Settings for two outputs with different signals

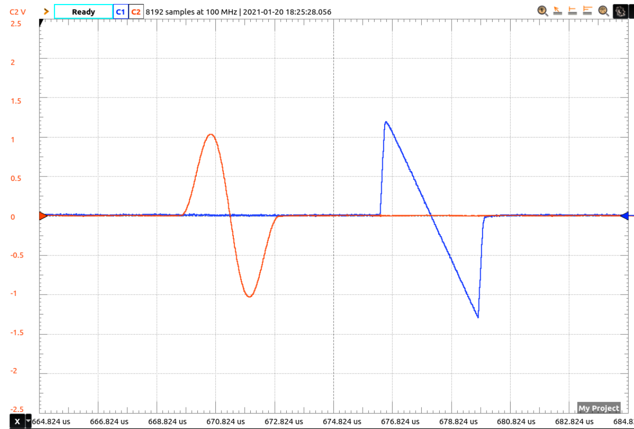

Figure 4.5: Two outputs with different signals

The advantages of windowing the sinusoidal pulse has been explained in Section 2.1.1. To showcase these advantages of using windowed sinusoidal pulses, the settings seen in Figure 4.6 have been used. First one can see the output signal in Figure 4.7. The FFT up to a frequency of 50 MHz has been computed for this signal in Figure 4.8, and up to 5 MHz in Figure 4.9 as well as up to 1 MHz in Figure 4.10. As a reference the FFT of the same 5 sinusoidal periods without the window has been plotted as well. Especially in the Figures showing lower frequencies, one can see that there are significantly less higher frequency components in the signal, as described in Section 2.1.1.

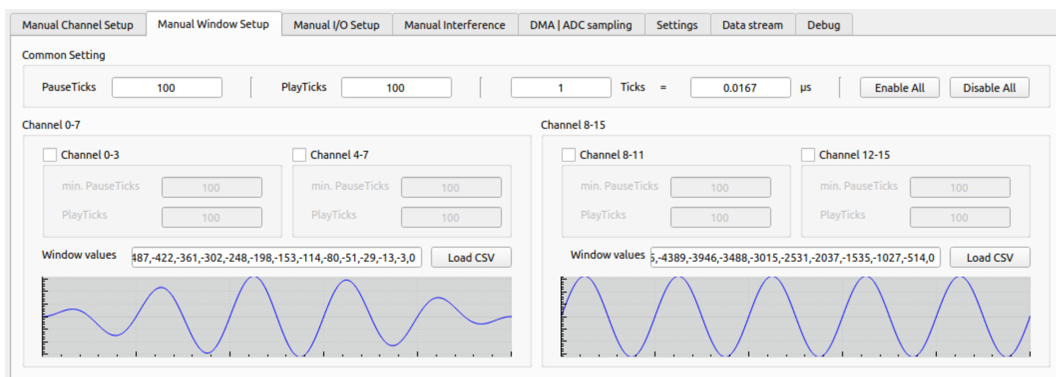

Figure 4.6: Settings for five periods of a sinusoidal signal multiplied by one cosine window

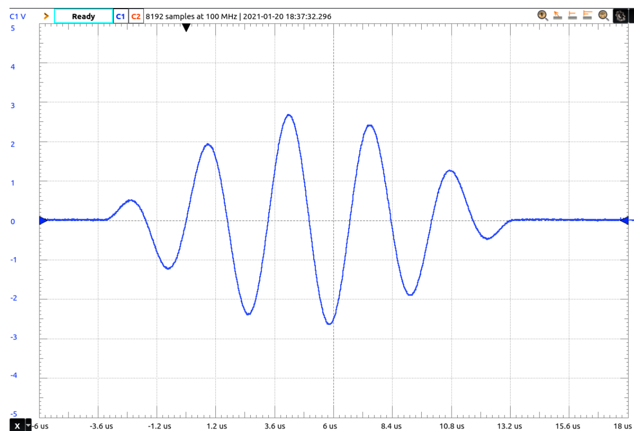

Figure 4.7: One pulse of five periods of a sinusoidal signal multiplied by one cosine window

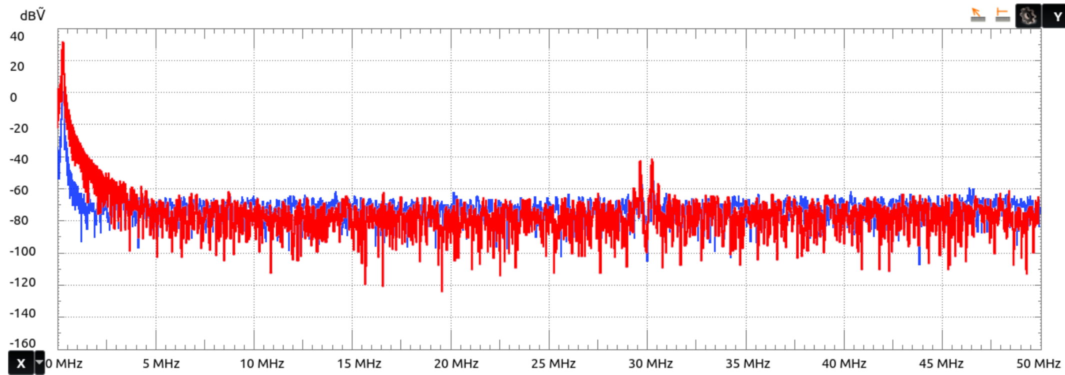

Figure 4.8: FFT of one pulse of five periods of a sinusoidal signal multiplied by one cosine window in blue and only the same 5 sinusoidal signals without windowing in red up to 50 MHz

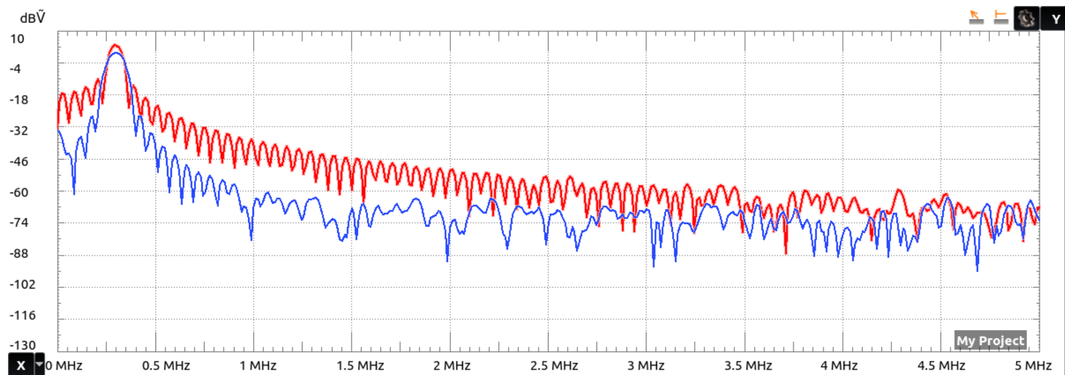

Figure 4.9: FFT of one pulse of five periods of a sinusoidal signal multiplied by one cosine window in blue and only the same 5 sinusoidal signals without windowing in red up to 5 MHz

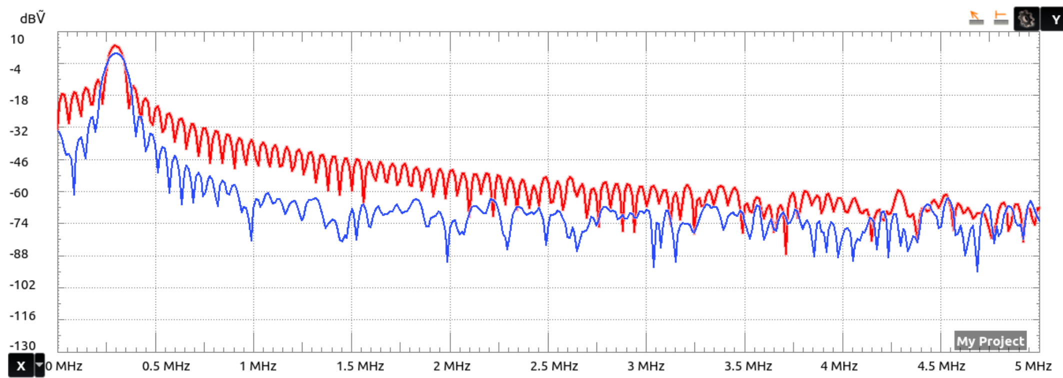

Figure 4.10: FFT of one pulse of five periods of a sinusoidal signal multiplied by one cosine window in blue and only the same 5 sinusoidal signals without windowing in red up to 1 MHz

5 Conclusion

This Master's Thesis aims to enhance the existing Flowmeter system, generating custom sinusoidal signals on 16 channels, with the possibility to generate as many different signal forms as possible for TOF measurement. This was achieved to the point of two different windows with four different channels regarding the delay in between the signal pulses.

It can further be said that the implementation of more custom signals using this method and hardware is not feasible, because the memory cannot be accessed fast enough, as thoroughly described in 3.3.3.1.

It is now possible to use this device for the further characterization of and the measurement on the desired application, for example SoH and SoC measurement of Li-ion batteries within the Mogli project, using the generation of windowed sine pulses, as seen in Section 4.

The final implementation is capable of generating arbitrary signals theoretically up to its clocking signal of 30 MHz, although a signal faster than 300 kHz is not feasible, as there would be less than 100 points per period. Additionally, a signal as slow as 114.4 Hz is possible due to the limits described in Section 3.4.2.3. This theoretical limit has been fully tested down to 14.6 kHz. The variable's length defining the pause in between signals is 32 bit, which means that a pause of over 143 s is possible.

Further the project has been packaged and documented thoroughly, hence the next person working on further improvements can start off with a substantial baseline and, especially, a virtual machine with everything installed and working as intended.

6 Outlook

As in every project there is always the possibility to improve the current state. In this case one practical addition would be to allow synchronization of the output signals, such that the phase difference in between different Delay Generators can be defined. This would significantly improve comparison of the effects of different signals.

Another improvement could be the implementation of more different output channels. Currently there are two different signals possible. The way this is implemented, each of the two different signals are sent to 4 DACs. As time multiplexing is already implemented in hardware, one can easily modify the FlowmeterControl application to generate a total of 8 different signals.

To gain even more different outputs, one could cache more data on the FPGA by for example transferring new data from memory only every second iteration of the signal.

Currently the transfer of window values from the FlowmeterControl application to the FlowmeterServer application is done by separating the values into individual integers and sending them one by one. The transfer will be significantly sped up, if all the values are sent together, as an array for example. To achieve this, a method to serialize arrays would need to be implemented.

At last one can try to find a way to start and stop the memory transfer from the PL side, to decrease the minimum pause time, which is caused because the Delay generator needs to wait for the start of the next pulse, as described in Section 3.3.4.1.

Acronyms

ADC Analog to Digital Converter. 1, 31, 34, 37
AIT Austrian Institute of Technology. 1
ARM® Advanced Reduced Instruction Set Computer Machine. 7–9, 23
ASIC Application Specific Integrated Circuit. 8
AVL Anstalt für Verbrennungskraftmaschinen List. 1
AXI Advanced eXtensible Interface. 9, 20, 21, 23, 37, 38, 40, 41

CISC Complex Instruction Set Machine. 7
CPLD Complex Programmable Logic Device. 8
CPU Central Processing Unit. 7

DAC Digital to Analog Converter. 1, 31, 34, 37, 39, 40, 42, 44, 59
DMA Direct Memory Access. 10, 20, 23–25, 37, 38, 40, 41, 47, 49
DSP Digital Signal Processor. 7
DTB Device Tree Blob. 36
DTS Device Tree Source. 36

EMS Institute of Electrical Measurements and Sensor Systems. iv, 1
EU European Union. 1

FFG Austrian Research Promotion Agency. iv, 1
FIFO First in First Out. 41
FPGA Field Programmable Gate Array. 2, 8, 9, 47, 59

GCC GNU Compiler Collection. 33
GPIO General Purpose Input/Output. 41, 50
GUI Graphical User Interface. 27, 47

HDF Hardware Description File. 45

I/O Input and/or Output. 8, 9, 30

IDE Integrated Development Environment. 9, 26
ILA Integrated Logic Analyzer. 20
IP Intellectual Property. 7, 18, 20, 21, 23, 31, 37–39, 41, 42, 44, 50

JSON JavaScript Object Notation. 27
JTAG Joint Test Action Group. 30, 31

Li-ion Lithium Ion. 5, 6, 58

MCU Microcontroller. 7
MM2S Memory to Stream. 24, 39, 40, 50

PCB Printed Circuit Board. 1
PL Programmable Logic. 9, 20, 23, 24, 27, 31, 41, 42, 46, 59
PLD Programmable Logic Device. 8, 9
PS Processing System. 9, 23, 24, 30, 37, 41

RAM Random Access Memory. 7
RISC Reduced Instruction Set Machine. 7, 8

S2MM Stream to Memory. 40
SDK Software Development Kit. 20, 36
SoC State of Charge. 5, 6, 58
SoF State of Fitness. 6
SoH State of Health. 6, 58
SoM System-on-Module. 1, 30

TDK TDK Corporation. 1
TOF Time of Flight. iv, v, 3, 4, 58
TOFD Time of flight diffraction ultrasonics. 4

VHDL Very High Speed Integrated Circuit Hardware Description Language. iv, v, 18, 20–22, 43, 44
VM Virtual Machine. 46

Bibliography

[1] Peter Hadley et al. *Phonon dispersion relation for a 1-d linear chain*. Jan. 28, 2021. URL: http://lampx.tugraz.at/~hadley/ss1/phonons/table/displc.html? (cit. on p. 4).

[2] Jasmin Blanchette and Mark Summerfield. "C++ GUI Programming with Qt 4, Second Edition." In: (Feb. 2015) (cit. on p. 26).

[3] RASSP E&F. *Basic VHDL*. 1995 (cit. on p. 21).

[4] Elizabeth Garnsey, Gianni Lorenzoni, and Simone Ferriani. "Speciation through entrepreneurial spin-off: The Acorn-ARM story." In: *Research Policy* 37.2 (2008), pp. 210–224. ISSN: 0048-7333. DOI: https://doi.org/10.1016/j.respol.2007.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0048733307002363 (cit. on pp. 7, 8).

[5] The Barr Group. *Embedded Systems Glossary*. Dec. 17, 2020. URL: https://barrgroup.com/Embedded-Systems/Glossary-E (cit. on pp. 7, 8, 10).

[6] Xilinx Inc. *Building and deploying applications with QT 4.8.5*. 2014. URL: https://www.xilinx.com/support/answers/59172.html (cit. on p. 31).

[7] Xilinx Inc. *IP Product Guide - AXI GPIO*. 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf (cit. on p. 40).

[8] Xilinx Inc. *PetaLinux Command Line Reference*. 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1157-petalinux-tools-command-line-guide.pdf (cit. on p. 44).

[9] Xilinx Inc. *Technical Reference Manual - Zynq-7000 SoC*. 2018. URL: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf (cit. on p. 35).

[10] Analog Devices Inc. *AD9707 Datasheet*. 2020. URL: https://www.analog.com/en/products/ad9707.html (cit. on p. 30).

[11] Avnet Inc. *PicoZed*. 2020. URL: http://zedboard.org/product/picozed (cit. on pp. 28, 29).

[12] Xilinx Inc. *AMBA AXI4 Interface Protocol*. 2020. URL: https://www.xilinx.com/products/intellectual-property/axi.html (cit. on p. 22).

[13] Xilinx Inc. *AXI DMA*. 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (cit. on pp. 36, 46).

[14] Xilinx Inc. *Behavioral Simulation*. Jan. 20, 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/platform_studio/ps_c_sim_timing_simulation.htm (cit. on p. 19).

[15] Xilinx Inc. *Creating and Packaging Custom IP*. 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1118-vivado-creating-packaging-custom-ip.pdf (cit. on p. 20).

[16] Xilinx Inc. *Platform Cable USB II Data Sheet*. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds593.pdf (cit. on p. 30).

[17] Xilinx Inc. *Structural Simulation*. Jan. 20, 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/platform_studio/ps_c_sim_structural_simulation.htm (cit. on p. 19).

[18] Xilinx Inc. *Timing Simulation*. Jan. 20, 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/platform_studio/ps_c_sim_behavioral_simulation.htm (cit. on p. 19).

[19] J. A. G. Temple J. P. Charlesworth. *Engineering Applications of Ultrasonic Time-of-Flight Diffraction Second Edition*. 2001. ISBN: ISBN 978-0-86380-239-3 (cit. on p. 4).

[20] R. Klambauer and A. Bergmann. "A new principle for an ultrasonic flow sensor for harsh environment." In: *2017 IEEE SENSORS*. 2017, pp. 1–3. DOI: 10.1109/ICSENS.2017.8234394 (cit. on pp. iv, v, 3, 28, 29).

[21] Purim Ladpli, Fotis Kopsaftopoulos, and Fu-Kuo Chang. "Estimating state of charge and health of lithium-ion batteries with guided waves using built-in piezoelectric sensors/actuators." In: *Journal of Power Sources* 384 (2018), pp. 342–354. ISSN: 0378-7753. DOI: https://doi.org/10.1016/j.jpowsour.2018.02.056. URL: http://www.sciencedirect.com/science/article/pii/S0378775318301770 (cit. on p. 6).

[22] Arm Limited. *ARM®Cortex™-A Series Programmer's Guide*. Jan. 8, 2020. URL: https://developer.arm.com/documentation/den0013/d/ (cit. on p. 7).

[23] Linuxconfig. *Debian vs Ubuntu*. 2021. URL: https://linuxconfig.org/debian-vs-ubuntu (cit. on p. 12).

[24] Terry O'Neal. *QT Build Instructions*. 2019. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842110/Qt+Qwt+Build+Instructions+Qt+5.4.2+Qwt+6.1.2 (cit. on p. 31).

[25] Adam Morawiec Philippe Coussy. *High-Level Synthesis - From Algorithm to Digital Circuit*. Springer, Dordrecht, 2008. ISBN: 978-1-4020-8588-8 (cit. on p. 19).

[26] H. Popp et al. "State Estimation Approach of Lithium-Ion Batteries by Simplified Ultrasonic Time-of-Flight Measurement." In: *IEEE Access* 7 (2019), pp. 170992–171000. DOI: 10.1109/ACCESS.2019.2955556 (cit. on pp. iv, v, 6).

[27] Hartmut Popp et al. "Mechanical Frequency Response Analysis of Lithium-Ion Batteries to Disclose Operational Parameters." In: *Energies* 11 (Mar. 2018), p. 541. DOI: 10.3390/en11030541 (cit. on pp. iv, v).

[28] Hartmut Popp et al. "Mechanical methods for state determination of Lithium-Ion secondary batteries: A review." In: *Journal of Energy Storage* 32 (2020), p. 101859. ISSN: 2352-152X. DOI: https://doi.org/10.1016/j.est.2020.101859. URL: http://www.sciencedirect.com/science/article/pii/S2352152X20316960 (cit. on p. 6).

[29] K.M.M. Prabhu. *Window Functions and Their Applications in Signal Processing*. Oct. 2013. ISBN: ISBN 9781466515833. DOI: 10.1201/9781315216386 (cit. on p. 5).

[30] Habiballah Rahimi Eichi et al. *Sensitivity Analysis of Lithium-Ion Battery Model to Battery Parameters*. Nov. 2013. DOI: 10.1109/IECON.2013.6700257 (cit. on p. 5).

[31] Florijan Reichmann. *Embedded System Design for a Time-of-Flight Ultrasonic Flowmeter*. Graz University of Technology, Austria. Nov. 2019 (cit. on pp. iv, v, 1–3, 9, 35, 44).

[32] tldp.org. *Bash Guide for Beginners - If statements*. 2021. URL: https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html (cit. on p. 16).

[33] Embedded Linux Wiki. *Device Tree Reference*. 2020. URL: https://elinux.org/Device_Tree_Reference (cit. on p. 35).

[34] Ubuntuusers Wiki. *Ubuntu Shells*. 2021. URL: https://wiki.ubuntuusers.de/Shell/ (cit. on p. 12).

[35] Wikipedia. *chmod*. Jan. 20, 2021. URL: https://en.wikipedia.org/wiki/Chmod (cit. on p. 17).